Metadata of the chapter that will be visualized in SpringerLink

Book Title	Soft Computing for Problem Solving	
Series Title		
Chapter Title	Long Short-Term Memory Recurrent Neural Network Architectures for Melody Generation	
Copyright Year	2019	
Copyright HolderName	Springer Nature Singapore Pte Ltd.	
Corresponding Author	Family Name	**Mishra**
	Particle	
	Given Name	**Abhinav**
	Prefix	
	Suffix	
	Role	
	Division	Machine Learning and Optimization Laboratory
	Organization	IIIT Allahabad
	Address	Allahabad, India
	Email	iim201401@iiita.ac.in
		iim2014003@iiita.ac.in
Author	Family Name	**Tripathi**
	Particle	
	Given Name	**Kshitij**
	Prefix	
	Suffix	
	Role	
	Division	Machine Learning and Optimization Laboratory
	Organization	IIIT Allahabad
	Address	Allahabad, India
	Email	icm2014007@iiita.ac.in
Author	Family Name	**Gupta**
	Particle	
	Given Name	**Lakshay**
	Prefix	
	Suffix	
	Role	
	Division	Machine Learning and Optimization Laboratory
	Organization	IIIT Allahabad
	Address	Allahabad, India
	Email	iit2014044@iiita.ac.in
Author	Family Name	**Singh**
	Particle	
	Given Name	**Krishna Pratap**
	Prefix	

Suffix	
Role	
Division	Machine Learning and Optimization Laboratory
Organization	IIIT Allahabad
Address	Allahabad, India
Email	kpsingh@iiita.ac.in

Abstract	Recent work in deep learning has led to more powerful artificial neural network designs, including recurrent neural networks (RNNs) that can process input sequences of arbitrary length. We focus on a special kind of RNN known as a long short-term memory (LSTM) network. LSTM networks have enhanced memory capability which helps them in learning sequences like melodies. This paper focuses on generating melodies using LSTM networks and conducting a survey for verifying quality of melody generated. We used the Nottingham ABC Dataset, which is a database of over 14,000 folk songs in ABC notation and serves as the training input for our RNN model. We have also conducted a Turing Test to give the quality of the music generated by the model. We will also discuss the overall performance, design of the model and adjustments made in order to improve performance.

Keywords (separated by '-')	Recurrent neural network - Long short-term memory - Sequential learning

Long Short-Term Memory Recurrent Neural Network Architectures for Melody Generation

Abhinav Mishra, Kshitij Tripathi, Lakshay Gupta and Krishna Pratap Singh

Abstract Recent work in deep learning has led to more powerful artificial neural network designs, including recurrent neural networks (RNNs) that can process input sequences of arbitrary length. We focus on a special kind of RNN known as a long short-term memory (LSTM) network. LSTM networks have enhanced memory capability which helps them in learning sequences like melodies. This paper focuses on generating melodies using LSTM networks and conducting a survey for verifying quality of melody generated. We used the Nottingham ABC Dataset, which is a database of over 14,000 folk songs in ABC notation and serves as the training input for our RNN model. We have also conducted a Turing Test to give the quality of the music generated by the model. We will also discuss the overall performance, design of the model and adjustments made in order to improve performance. AQ1

Keywords Recurrent neural network · Long short-term memory
Sequential learning

A. Mishra (✉) · K. Tripathi · L. Gupta · K. P. Singh
Machine Learning and Optimization Laboratory, IIIT Allahabad, Allahabad, India
e-mail: iim201401@iiita.ac.in; iim2014003@iiita.ac.in

K. Tripathi
e-mail: icm2014007@iiita.ac.in

L. Gupta
e-mail: iit2014044@iiita.ac.in

K. P. Singh
e-mail: kpsingh@iiita.ac.in

J. C. Bansal et al. (eds.), *Soft Computing for Problem Solving*, Advances in Intelligent Systems and Computing 817, https://doi.org/10.1007/978-981-13-1595-4_4

1 Introduction

In recent years, neural networks have become widely popular and are often mentioned along with terms such as machine learning, deep learning, data mining and big data. Deep learning methods perform better than traditional machine learning approaches on virtually every single metric. From Google's DeepDream that can learn an artists style, to AlphaGo learning an immensely complicated game as Go, the programs are capable of learning to solve problems in a way our brains can do naturally. To clarify, deep learning, first recognized in the 80s, is one paradigm for performing machine learning. Unlike other machine learning algorithms that rely on hard-coded feature extraction and domain expertise, deep learning models are more powerful because they are capable of automatically discovering representations needed for detection or classification based on the raw data they are fed.

Artificial Neural networks are biologically inspired paradigm which enables a machine to learn from observational data. They are very robust and powerful learning models that have resulted in the best and trendsetting results in variety of supervised and unsupervised problems of machine learning and data science. This success has been possible only due to the ability of these models to learn hierarchical representation. These models are able to extract the underlying patterns that have not been possible with other algorithms [1].

Recurrent neural networks (RNNs) are connectionist models. RNNs have the capability to pass information which are relevant across the sequence steps as they process the sequential data elements one at a time. RNNs are able to model sequential set of input to the corresponding set of outputs that may be dependent on other input element or set of input elements.

Why recurrent neural networks are useful and why it's worth inspecting? It is a question that we will try to answer in the following subsections. We are focused and motivated by the desire to achieve empirical results. RNNs have its root in both cognitive modelling and machine learning, but we will focus on achieving the empirical result. Many foundational papers [2–4] have devalued the biological inspiration in favour of achieving empirical results on various problems. Before diving into the architecture of LSTM networks, we will begin by the architecture of a regular recurrent neural network and its issues, and how LSTMs resolve that issue.

2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of neuron-based models for processing continuous and interdependent sequence. Various architectures of neural network are specialized in solving different types of problems. For example, convolutional neural networks are specialized in processing and finding hidden patterns in images; recurrent neural network on the other hand is suited for other group of tasks. RNNs are best suited for tasks in which we have to derive a relationship from a sequence of values $x^{(1)}, \dots, x^{(\tau)}$ and the corresponding output $y^{(1)}, \dots, y^{(\tau)}$. RNNs are same as feedforward neural network, only difference being the edges that span along time steps. These edges are called recurrent edges. These cycles are responsible for the interdependency of the present value of a node on its value at next or upcoming time step and brings a notion of time to the model. RNNs can scale to much longer sequences which is not possible by a model without sequence-based specialization. RNNs are also capable of processing sequences of variable lengths. We usually apply recurrent neural network by operating on mini-batches. The time step index t used in the input vector $x^{(t)}$ generally not refers to the notion of the time and may refer the successive positions in the sequential input. A well known result was proposed by Siegelman and Sontag in 1991 that a recurrent neural network having a finite size with the sigmoid as activation function can simulate universal Turing machine [5].

There are various ways possible for building a RNN. Recurrent neural network has the properties that allows it to model any function which involves a previous time step dependence or recurrence where as feedforward networks cannot model functions where there is a notion of time or recurrence involved. The following equation is widely used by RNNs to define the values of their hidden units.

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, x^{(t)}; \theta)$$

There are some additional architectural advancements such as output layers that use state information $\boldsymbol{h}^{(t)}$ to predict the output $\hat{\boldsymbol{y}}^{(t)}$ at the tth time step. The network learns to use $\boldsymbol{h}^{(t)}$ as a kind of compact summary of the task-relevant aspect of the already processed sequence which allows the network to perform task that requires prediction of future from the available information in the past. The network makes a mapping of the processed sequence $(x^{(t)}, x^{(t-1)}, \dots, x^{(2)}, x^{(1)})$ to a arbitrary lengthed vector $\boldsymbol{h}^{(t)}$. There is a possibility that the relevant information from past stored in $\boldsymbol{h}^{(t)}$ might contain some features of the past with more precision and some it may forget depending on the training criterion. Figure 1 depicts a simple recurrent neural network, and we try to develop the equations for the forward pass. A loss function $\boldsymbol{L}$ represents the negative log-likelihood which gives the measurement of how far each $o^{(t)}$ is from the corresponding $y^{(t)}$. The training of the recurrent neural network is done in two passes similar to feedforward neural network, forward pass and backward pass. Following equations specify all the computational calculations at different time steps during the forward pass.

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \text{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

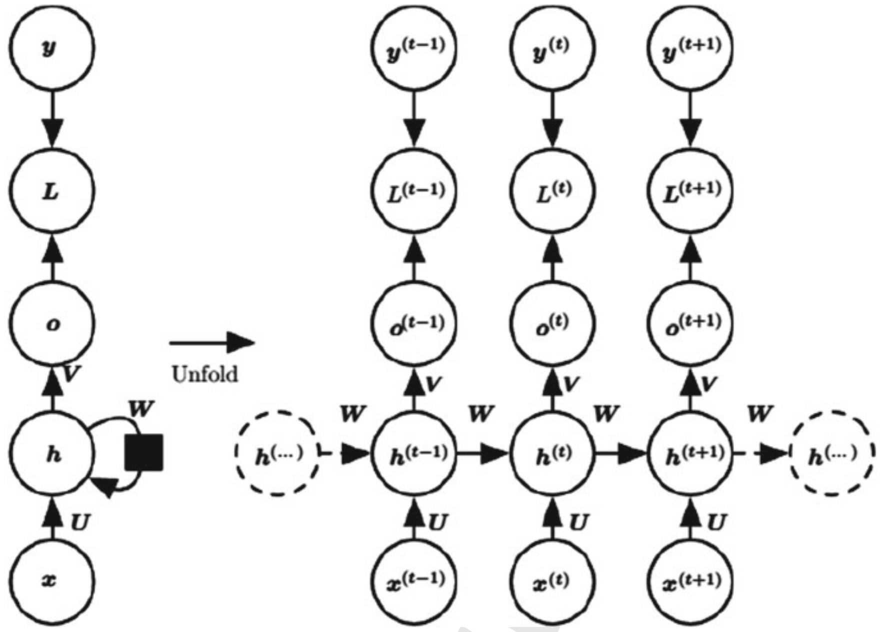


Fig. 1 Recurrent neural network unfolded for the each time step to compute the training loss. Input sequence $x^{(t)}$ are mapped to corresponding sequence of output values $\hat{y}^{(t)}$

Here, the connections between the layers of RNNs are parametrized by weight matrices. The connections between input to hidden layer are parametrized by $\boldsymbol{U}$, connections between hidden to hidden layer by $\boldsymbol{W}$ and connections between hidden to output layer by $\boldsymbol{V}$. $\boldsymbol{b}$ and $\boldsymbol{c}$ represent the bias parameters which is necessary for the model to learn the offset. AQ2

L represents the overall cost for a continuous sequence of $\boldsymbol{x}$ values along with a continuous sequence of $\boldsymbol{y}$ values which will be the aggregate of all the individual losses over all the time steps. Given $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$, the negative log-likelihood of $y^{(t)}$ is given by $L^{(t)}$ as

$$L\Big(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\Big) = \sum_t L^{(t)}$$

$$= -\sum_t \log p_{model}\Big(y^{(t)}|\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\Big),$$

where the value of $p_{model}\big(y^{(t)}|\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\big)$ is determined by the corresponding entry of $y^{(t)}$ in the output vector of t time step $\hat{\boldsymbol{y}}^{(t)}$. We need to compute the gradient of the loss function with respect to the parameters in order to train the model. The gradient calculation is computationally expensive operation and cannot be reduced by parallel computation due to the inherent sequentiality of the nodes. The network can be trained across many time steps using backpropagation. The expansion of backpropagation across time steps is called *backpropagation through time* (BPTT) [6]. The run-time complexity and the memory complexity of the forward pass is $O(\tau)$.

2.1 Training Recurrent Networks

Training with recurrent networks has always been considered a difficult task. Even for a single-layered feedforward networks, the optimization task is NP-complete [7] and learning the long-range dependencies makes the job of learning with recurrent network even more difficult. For training the RNN, we need to compute the gradient of loss function with respect to parameters. For this, we simply apply the generalized backpropagation algorithm to the unfolded RNN across time steps. Gradient resulted by backpropagation can be employed with different gradient-based training technique such as Gradient Descent, Adagrad. The nodes of our RNN include parameters $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}$ and $\boldsymbol{c}$. It also includes the respective input $\boldsymbol{x}^{(t)}$, hidden state $\boldsymbol{h}^{(t)}$, output preactivation $\boldsymbol{o}^{(t)}$ and the loss function $\boldsymbol{L}^{(t)}$ at the t time step. The gradient computation of each node $\mathbf{N}$, $\nabla_{\mathbf{N}}L$ is required to be done recursively, which depend on the gradient of precomputed nodes. The recursion begins with the nodes preceding the final loss.

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

In the following equations, we will assume that we are using softmax function for calculating the output probability vector $\hat{\boldsymbol{y}}^{\boldsymbol{t}}$ from output $\boldsymbol{o}^{(t)}$. We also assume that the loss is the cross-entropy loss. The gradient with respect to the outputs $\boldsymbol{o}^{(t)}$ at the time step t is

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial {o_i}^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial {o_i}^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}$$

The gradient calculation begins from the end of the sequence. The gradient calculation for $\boldsymbol{h}^{(t)}$ depends on whether t is the final time step. If $t = \tau$, where τ is the final time step, gradient of loss function with respect to $\boldsymbol{h}^{(\tau)}$ is:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = V^{\top} \nabla_{\boldsymbol{o}^{(\tau)}} L.$$

From $t = \tau - 1$, the gradient is then propagated backwards in time to $t = 1$. Since both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$ are descendents of $\boldsymbol{h}^{(t)}$, hence its gradient is computed as

$$(\nabla_{\boldsymbol{h}^{(t)}} L) = \frac{\partial L}{\partial h^{(t)}} = \frac{\partial L}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} + \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}}$$

$$= \boldsymbol{W}^{\top}(\nabla_{\boldsymbol{h}^{(t+1)}} L) diag\Big(1 - \big(\boldsymbol{h}^{(t+1)}\big)^2\Big) + \boldsymbol{V}^{\top}(\nabla_{\boldsymbol{o}^{(t)}} L)$$

As the gradients of the loss function with respect to the internal nodes are calculated, gradients on the parameter nodes are obtained.

2.2 Challenges of Long-Term Dependencies

The major difficulty that optimization algorithms like backpropagation through time (BPTT) faces when the depth of network increases too much. Depth of the network represents the sequence length or the time sequence duration. When deep computational graphs are constructed by repeatedly applying the same operation at each time step of a long temporal sequence, it gives rise to noticeable difficulties. We are able to see that activation function such as $tanh$ (Fig. 2) and $sigmoid$ tend to approach a flat line when value of the input activation function increases too much, resulting in the derivatives limiting to zero at both ends and the saturation of the corresponding neurons. The neurons have a zero gradient, and it results in driving other gradients in previous layers towards zero. Hence, the multiple matrix multiplications result in exponential shrinking of the gradient values, vanishing after few time steps only. Since gradient contributions of steps which are far become negligible, it does not contribute to learning procedure making the model unable to learn the long-term dependencies. This problem is called vanishing gradient problem [8], and it is not

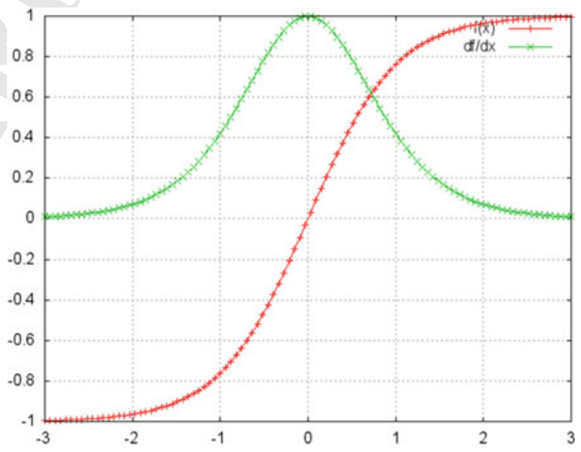


Fig. 2 The red line represents the gradient of *tanh(x)*. The gradient reaches zero as |x| takes large value

exclusive only to RNNs and may occur in all connectionist models in which depth of layers increases.

It is possible to encounter exploding gradients instead of vanishing gradients hinging on the activation functions and the architecture of the network. There is a twofold reason as to why it is vanishing gradient problem only that is more talked about than exploding gradient problem. First one being that if the gradient becomes two large it will NaN (not a number). The second reason is that gradients can be clipped after a defined threshold. Vanishing gradients are not this easy to tackle, and unlike exploding gradients it is not obvious when this problem may occur.

The vanishing gradient problem can be easily tackled. Regularization and initialization of weight matrices with proper weights are two ways to combat the problem. A more better and commonly preferred solution is the usage of Rectifying Linear Unit (ReLU) activation function instead of using $sigmoid$ or $tanh$ activation functions. ReLU allows only two possible gradient values 0 and 1, which makes it less susceptible to vanishing gradient problem. Other possible solution for this results in change in basic architecture of recurrent neural network architecture. *Long short-term memory* (LSTM) and *gated recurrent unit* (GRU) [9] architectures are advancements in vanilla RNN architecture which has resulted in better learning and long-range dependencies. LSTMs are one of the most widely used models in Natural Language Processing and were proposed in 1997. Both LSTM and GRU architectures are designed explicitly for dealing with the problem of vanishing gradient and for learning long-range dependencies. Improved architectures and better gradient-following heuristics have made training of RNN possible and feasible. Packages and libraries like *Theano*, *Torch* and *Tensorflow* have made possible the efficient implementation of challenging architectures of multilayered neural networks.

3 LSTM Architecture

The vanilla RNN architecture suffers from the various complications like vanishing gradient problem, convergence problems. There was need of more robust architectures that do not suffer same complications as RNN. In 1997, two research papers published gave the most successful recurrent neural network architecture for sequence modelling. The first paper, *Long Short-Term Memory* (LSTM) model [10], proposes a brilliant idea of introduction of self-loops in order to produce paths where the flow of gradient for long duration is possible. A memory cell is introduced which acts as a unit of computation and replaces the traditional hidden nodes of the network.

3.1 Long Short-Term Memory (LSTM)

The LSTM model was introduced by Hochreiter and Schmidhuber in 1997. LSTMs are explicitly designed to tackle the problem of long-term dependencies. LSTM-RNN is a type of gated recurrent neural network. Gated RNNs are able to make

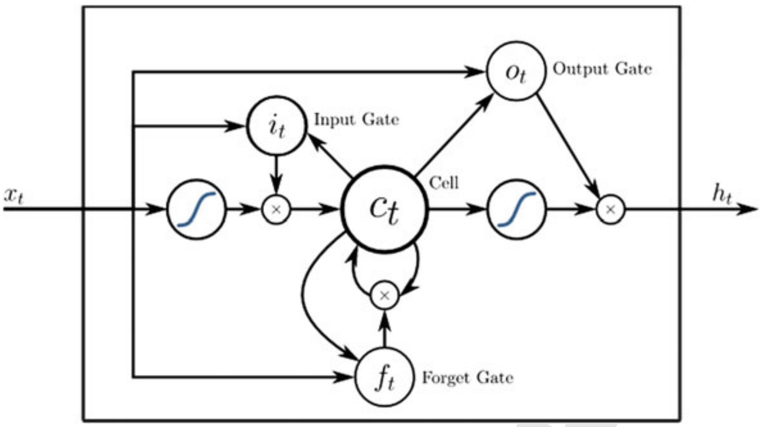


Fig. 3 A LSTM block with input, output and forget gates

paths through time whose derivatives neither explode nor vanish. Rather than being fixed, the weight of the self-loop is conditioned on the context. The structure of RNNs is like a chain of repeated modules. In vanilla RNNs, this repeating module is something as simple as a *tanh* layer, however, in LSTMs [11] this chain-like structure comprises of module having different architecture. In this architecture, there are four neural network layers instead of one, interacting in a very special way. Machine translation, unconstrained handwriting recognition [12], handwriting generation, protein sequence prediction [13], image captioning and parsing, etc., are some of the applications in which LSTM has proved to give groundbreaking results and has been used recurrently by the scientific community.

The LSTM block diagram is illustrated in Fig. 3. LSTMs contain LSTM cells having recurrence relation internally instead of traditional nodes in vanilla RNN that simply apply a element wise nonlinearity of inputs and recurrent units. Each of these cells works as a ordinary recurrent network, and there is a system of gated units which control the flow of information.

The most crucial part of the state unit here is $s_i^{(t)}$ that contains a linear self-loop; however, the controlling of the self-loop weights is done by a **forget gate** unit f_i^t for the ith cell and tth time step. This unit sets the weight of the self-loop to a value between 0 and 1 using activation function.

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right),$$

where $\boldsymbol{x}^{(t)}$ represents input vector and $\boldsymbol{h}^{(t)}$ represents hidden layer at t time step. $\boldsymbol{b}^f$, $\boldsymbol{U}^f$, $\boldsymbol{W}^f$ represent biases, weights and recurrent weights for the forget gates, respectively. Internal states are updated as follows:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right),$$

Here $\boldsymbol{b}$, $\boldsymbol{U}$, $\boldsymbol{W}$ represent the biases, input weights and recurrent weights in the LSTM cell, respectively. $g_i^{(t)}$ is called **external input gate** unit and is computed the same way as the forget gate. It gives a gating value between 0 and 1 for the current input value and uses its own parameters $\boldsymbol{b}^g$, $\boldsymbol{U}^g$, $\boldsymbol{W}^g$:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right),$$

There is also a **output gate** $q_i^{(t)}$ which controls the output $h_i^{(t)}$ of the LSTM and uses sigmoid unit for gating:

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right)$$

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$$

Here $\boldsymbol{b^o}$, $\boldsymbol{U^o}$, $\boldsymbol{W^o}$ denote the biases, input weights and recurrent weights for the output gate of the LSTM cell.

Researches have been able to achieve exceptional results and accuracies in variety of applications using RNNs and fare share of those is achieved using LSTMs only. Several models and variations have been proposed based on long short-term memory model. The Forget gates were not proposed in the original LSTM design, and it was later proposed in 2000. Use of forget gates have been effective and hence are used as a standard design in the implementation. Grid LSTM by Kalchbrenner [14] also seems extremely promising architecture. LSTM was a big step in what we can achieve and accomplish with RNNs and a great possibility of future work. This field of machine learning has seen a tremendous run in the last few years and coming ones promise to only be more so.

4 Overview of Training Process

Learning problems can be grouped into two basic categories: supervised and unsupervised. Supervised learning includes classification, prediction and regression, where the input vectors have a corresponding target (output) vectors. The goal is to predict the output vectors based on the input vectors. In unsupervised learning, such as clustering, there are no target values and the goal is to describe the associations and patterns among a set of input vectors [15]. The LSTM implementation solves a supervised learning problem: given a sequence of inputs, we want to predict the probability of the next output.

Normally to perform machine learning, it is best to break a given dataset into three parts: a training set, a validation set and a test set. The training set is used for learning; the validation set is used to estimate the prediction error for model selection; the test set is used for assessment of the generalization error of the final chosen model. A general rule of thumb is to split the dataset 50% for training and 25% each for validation and testing.

In our case, our goal is to build a model that can predict the next word or that next music note; this is a generative model in which we can generate new text or music by sampling from the output probabilities. Therefore, we will be having training and validation set to fine tune the model, but we will not be having a test set. Instead, to generate an output we can feed in a randomly selected batch of data from the training.

5 Implementation of Melody Generation

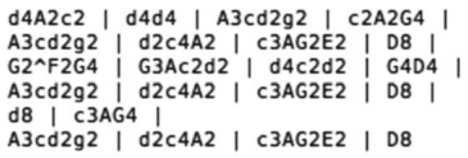
5.1 Char-RNN Model

The char-RNN model is a robust model which uses LSTM architecture to learn the structure of the input in the form of text file. It takes a single text file as an input and feeds it into the RNN algorithm that learns to predict the next character in the sequence. After training the RNN, it can generate text character by character that looks stylistically similar to the original dataset. The code is written in Python and uses Numpy library. In this model, we have option to have multiple layers, supporting code for model checkpointing, and using mini-batches to make the learning process efficient.

5.2 Input Dataset: ABC Notation

We used Nottingham Folk Music Dataset as input for the char-RNN model. It is a collection of tunes in ABC format. Nottingham Music Dataset is a combination of 14,000 British and American folk tunes. The ABC notation was developed by Chris Walshow so that music can be represented using ASCII symbols [16]. The basic structure of the abc notation is the header and the notes. The header which contains background information about the song can look something like below:

```
X: 145
T: A and A's Waltz
S: Mick Peat
M: 6/8
K: D
```



```
d4A2c2 | d4d4 | A3cd2g2 | c2A2G4 |
A3cd2g2 | d2c4A2 | c3AG2E2 | D8 |
G2^F2G4 | G3Ac2d2 | d4c2d2 | G4D4 |
A3cd2g2 | d2c4A2 | c3AG2E2 | D8 |
d8 | c3AG4 |
A3cd2g2 | d2c4A2 | c3AG2E2 | D8
```

Fig. 4 ABC notation

These lines are known as fields, where X is the reference number, T is the title of the song, C is the composer, M is the metre, and K is the key. The notes portion looks something like in Fig. 4.

6 Results and Discussion

6.1 Checkpoints and Minimum Validation Loss

During training, a checkpoint is saved every 1,000 iterations and at every checkpoint. The checkpoint file saves the current values for all the weights in the model. The smaller the loss, the better the checkpoint file works when generating the music. Due to possible overfitting, the minimum cross-entropy validation loss is not necessarily at the end of the training. For example, the Table 1 and Fig. 5 show all the saved checkpoints during a single training. AQ3

The minimum validation loss occurs at the 7,000th iteration out of 10,000 iterations. Rerunning the code on the same dataset produced the same loss values.

Table 1 Iteration versus cross-entropy loss

Nth iteration (out of 10000)	Cross-entropy validation loss
1000	381.34
2000	216.56
3000	131.21
4000	80.75
5000	73.20
6000	71.92
7000	69.32
8000	72.64
9000	74.21
10,000	86.52

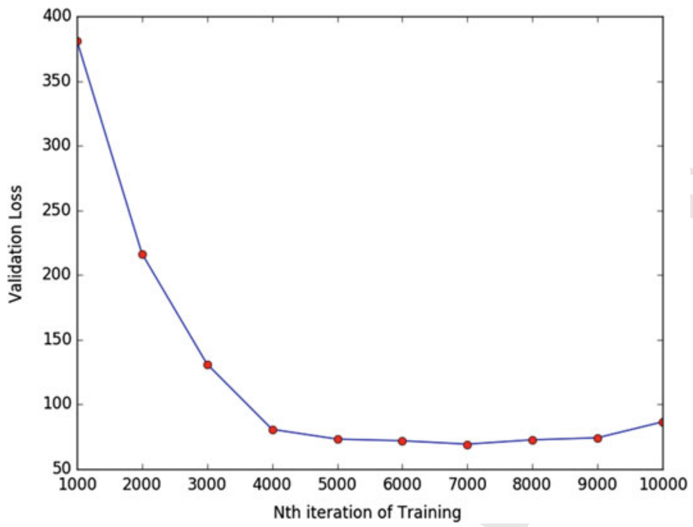


Fig. 5 Plot of validation losses for various checkpoints saved at every 1000th iteration

Each iteration on average takes about 0.33 s, and in total the model takes about 3–4 h to train on CPU.

6.2 *Decreasing the Batch Parameter*

These default parameters worked well for the dataset. As an experiment, we decreased the batch size from the default size of 100–50. The batch size specifies how many streams of data are processed in parallel at one time. If the input text file has N characters, these get split into chunks of size [batch size] $\times$ [sequence length] (length

Table 2 Iteration versus cross-entropy loss (batch size = 50)

Nth iteration (out of 10,000)	Cross-entropy validation loss
1000	411.32
2000	226.65
3000	121.38
4000	85.25
5000	73.61
6000	66.62
7000	69.82
8000	74.41
9000	74.28
10,000	79.52

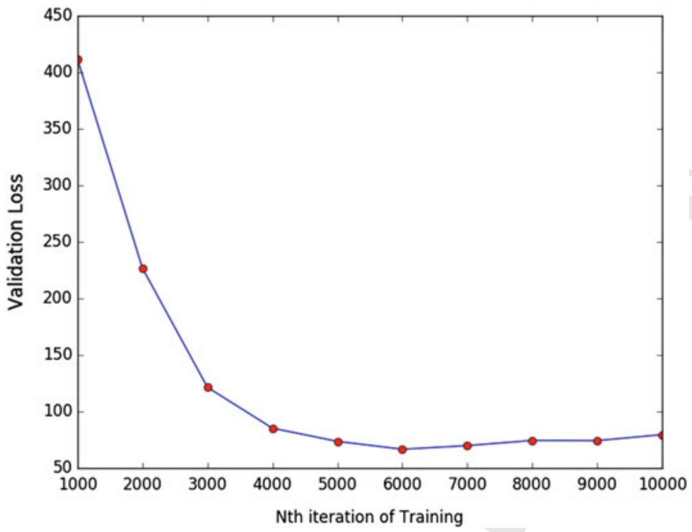
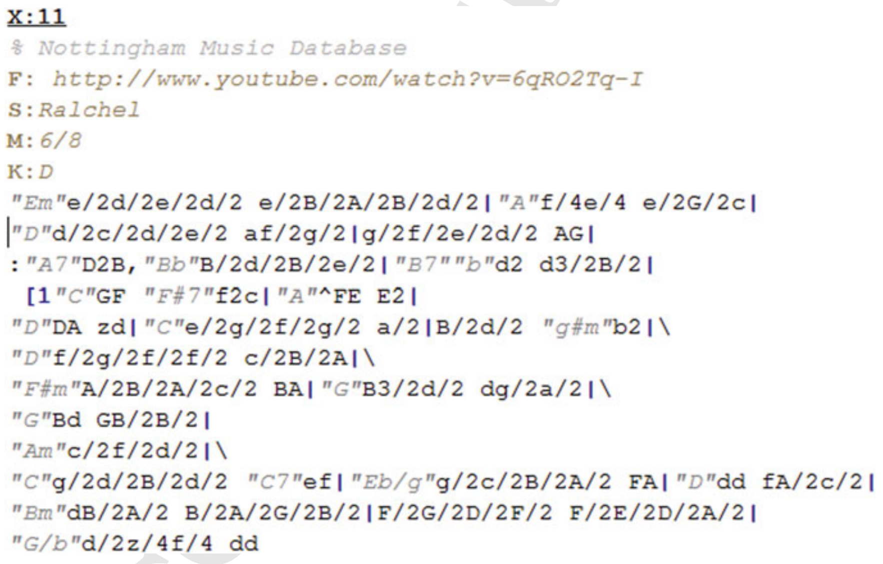


Fig. 6 Plot of validation losses for various checkpoints saved at every 1000th iteration where batch size = 10,000



```
X:11
% Nottingham Music Database
F: http://www.youtube.com/watch?v=6qRO2Tq-I
S:Ralchel
M:6/8
K:D
"Em"e/2d/2e/2d/2 e/2B/2A/2B/2d/2|"A"f/4e/4 e/2G/2c|
|"D"d/2c/2d/2e/2 af/2g/2|g/2f/2e/2d/2 AG|
:"A7"D2B,"Bb"B/2d/2B/2e/2|"B7""b"d2 d3/2B/2|
 [1"C"GF "F#7"f2c|"A"^FE E2|
"D"DA zd|"C"e/2g/2f/2g/2 a/2|B/2d/2 "g#m"b2|\
"D"f/2g/2f/2f/2 c/2B/2A|\
"F#m"A/2B/2A/2c/2 BA|"G"B3/2d/2 dg/2a/2|\
"G"Bd GB/2B/2|
"Am"c/2f/2d/2|\
"C"g/2d/2B/2d/2 "C7"ef|"Eb/g"g/2c/2B/2A/2 FA|"D"dd fA/2c/2|
"Bm"dB/2A/2 B/2A/2G/2B/2|F/2G/2D/2F/2 F/2E/2D/2A/2|
"G/b"d/2z/4f/4 dd
```

Fig. 7 Sample output text

of each step, which is also the length at which the gradients can propagate backwards in time).

The resulting validation loss was less than when trained with batch size of 50, as shown in the Table 2 and Fig. 6.

Figure 7 is a sample output text from the model (default parameters).

Table 3 Confusion matrix

	Machine-generated	Human-generated	
Machine-generated	611	889	40.7%
Human-generated	233	1267	84.4%
	72%	58%	62.6%

6.3 Turing Test

We collected a set of 15 artificially produced melodies by our model and 15 human-generated melodies and took a survey to classify the music as human-generated or machine-generated on a set of 100 people. The results of the survey are represented as confusion matrix as follows.

As per the Turing Test, it was found that people were not able to distinguish between the human-generated and machine-generated melodies. After conducting the test, we found that over 62% of the test subjects were convinced that the melodies were at par with human music.

The challenge of getting a successful result from this char-RNN model was the limitation of the small dataset. There are not many songs that are in abc format, and the songs themselves are short (only about 10–20 measures per song) and very simple tunes. The first time we ran the char-RNN it was on a dataset of 277 KB, which is far less than the minimum required size of 1 MB for the char-RNN to produce tangible results. After finding more music from another source, the dataset increased to 451 KB, and the results were significantly better. Overall, music generated from the model with the larger dataset sounded better (Table 3).

7 Conclusion

The LSTM-RNN is able to learn from the songs and generate melodies on its own which sounds similar to human made music. However, there are some outputs where occasionally there would be a note or two that does not sound cohesive with the rest of the music, which may be because the note is not part of the music's scale. LSTM models have proven to be robust in generating music that is very similar to the given input dataset.

There are many optimization techniques involved in neural networks that people may explore further. Overall we are impressed with the progress and results deep learning techniques have accomplished in the recent years. It is fascinating to see deep neural networks' capability to create something that is aligned with the style of the given input, and yet have its own unique taste that can be sometimes bizarre but sometimes incredible. Researchers may collaborate with people with background in music theory in order to improve the accuracy of the model.

References

1. Karpathy, A.: The Unreasonable Effectiveness of Recurrent Neural Networks, Github, 21 May 2015, Web 04 May 2016
2. Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE Trans. Signal Process. **45**(11), 2673–2681 (1997)
3. Socher, R., Karpathy, A., Le, Q.V., Manning, C.D., Andrew, Y.Ng.: Grounded compositional semantics for finding and describing images with sentences. Trans. Assoc. Computat. Linguist. **2**, 207–218 (2014)
4. Karpathy, A., Fei-Fei, L.: Deep Visual-Semantic Alignments for Generating Image Descriptions (2014). arXiv:1412.2306
5. Siegelmann, H.T., Sontag, E.D.: Turing computability with neural nets. Appl. Math. Lett. **4**(6), 77–80 (1991)
6. Werbos, P.J.: Backpropagation through time: what it does and how to do it. Proc. IEEE **78**(10), 1550–1560 (1990)
7. Blum, A.L., Rivest, R.L.: Training a 3-node neural network is NPcomplete. In: Machine Learning: From Theory to Applications, pp. 9–28. Springer (1993)
8. Hochreiter, S.: The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int. J. Uncertain. Fuzziness Knowl. Based Syst. (1995)
9. Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. **18**(5), 602–610 (2005)
10. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
11. Gers, F.A.: Long short-term memory in recurrent neural networks. Unpublished Ph.D. dissertation, Ecole Polytechnique Federale de Lausanne, Lau sanne, Switzerland, 2001
12. Graves, A., Liwicki, M., Bunke, H., Schmidhuber, J., FernÃandez, S.: Unconstrained on-line handwriting recognition with recurrent neural networks. In: Platt, J., Koller, D., Singer, Y., Roweis, S. (eds.) NIPS’2007, pp. 577–584 (2008)
13. Baldi, P., Brunak, S., Frasconi, P., Soda, G., Pollastri, G.: Exploiting the past and the future in protein secondary structure prediction. Bioinformatics **15**(11), 937–946 (1999)
14. Kalchbrenner, N., Danihelka, I., Graves, A.: Grid Long Short-Term Memory (2015). arXiv:1507.01526
15. Graves, A.: Supervised Sequence Labelling with Recurrent Neural Networks, vol. 385. Springer (2012)
16. Walshaw, C.: How to Understand Abc (the Basics). Web log post. ABC Notation Blog. WordPress, 23 Dec 2009, Web 18 Dec 2015

Author Queries

Chapter 4

Query Refs.	Details Required	Author's response
AQ1	Please check and confirm if the author names and initials are correct.	
AQ2	Please suggest whether the phrase "hidden to hidden layer" can be retained as such.	
AQ3	Please check and confirm if the inserted citation of Tables 1–3 are correct. If not, please suggest an alternate citation. Please note that tables should be cited sequentially in the text.	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	• • • under matter to remain	✓⃝
Insert in text the matter indicated in the margin	⋏	New matter followed by ⋏ or ⋏ⓧ2
Delete	/ through single character, rule or underline or ⊢—⊣ through all characters to be deleted	ꝺ/ or ꝺ/ⓧ2
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢—⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌣
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	~ under matter to be changed	~
Change to bold italic	≂ under matter to be changed	≂
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⊬
Change bold to non-bold type	(As above)	⫮
Insert 'superior' character	/ through character or ⋏ where required	⋎ or ⋏ under character e.g. $\overset{2}{\curlyvee}$ or $\overset{2}{\curlywedge}$
Insert 'inferior' character	(As above)	⋏ over character e.g. $\underset{2}{\curlywedge}$
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	$\overset{‘}{\curlyvee}$ or $\overset{‘}{\curlywedge}$ and/or $\overset{’}{\curlyvee}$ or $\overset{’}{\curlywedge}$
Insert double quotation marks	(As above)	$\overset{“}{\curlyvee}$ or $\overset{“}{\curlywedge}$ and/or $\overset{”}{\curlyvee}$ or $\overset{”}{\curlywedge}$
Insert hyphen	(As above)	⊢⊣
Start new paragraph	_⌐	_⌐
No new paragraph	⤶	⤶
Transpose	⊔⊓	⊔⊓
Close up	linking ◠◡ characters	◠◡
Insert or substitute space between characters or words	/ through character or ⋏ where required	Y
Reduce space between characters or words	| between characters or words affected	⊤