Long Short-Term Memory Recurrent Neural Network Architectures for Generating Music and Japanese Lyrics

Ayako Mikami
2016 Honors Thesis
Advised by Professor Sergio Alvarez
Computer Science Department, Boston College

Abstract

Recent work in deep machine learning has led to more powerful artificial neural network designs, including Recurrent Neural Networks (RNN) 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, creating the possibility of using them for learning and generating music and language.

This thesis focuses on generating Chinese music and Japanese lyrics using LSTM networks. For Chinese music generation, an existing LSTM implementation is used called char-RNN written by Andrej Karpathy in the Lua programming language, using the Torch deep learning library. I collected a data set of 2,500 Chinese folk songs in abc notation, to serve as the LSTM training input. The network learns a probabilistic model of sequences of musical notes from the input data that allows it to generate new songs. To generate Japanese lyrics, I modified Denny Britz's GRU model into a LSTM networks in the Python programming language, using the Theano deep learning library. I collected over 1MB of Japanese Pop lyrics as the training data set. For both implementations, I discuss the overall performance, design of the model, and adjustments made in order to improve performance.

Contents

1. Introduction

1.1 Overview

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 artist's 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 80's, 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. [13] For this thesis, we focus on a type of machine learning technique known as artificial neural networks. When we stack multiple hidden layers in the neural networks, they are considered deep learning. Before diving into the architecture of LSTM networks, we will begin by studying the architecture of a regular neural network, then touch upon recurrent neural network and its issues, and how LSTMs resolve that issue.

1.2 Feedforward Neural Networks

Neural network is a machine learning technique inspired by the structure of the brain. The basic foundational unit is called a neuron. Every neuron accepts a set of inputs and each input is given a specific weight. The neuron then computes some function on the weighted input. Functions performed throughout the network by the neurons include both linear and nonlinear – these nonlinear functions are what allow neural networks to learn complex nonlinear patterns. Nonlinear functions include sigmoid, tanh, ReLU, and Elu; these functions have relatively simple derivatives, which is an important characteristic that will be discussed later in this section. Whatever value the functions computes from the weighted inputs are the outputs of the neuron that are then transmitted as inputs to succeeding neuron(s). The connected neurons then form a network, hence the name neural network. The basic structure of a neural network consists of three types of layers: input layer, hidden layer, and output layer. The diagram below is an example of a neural network's structure.

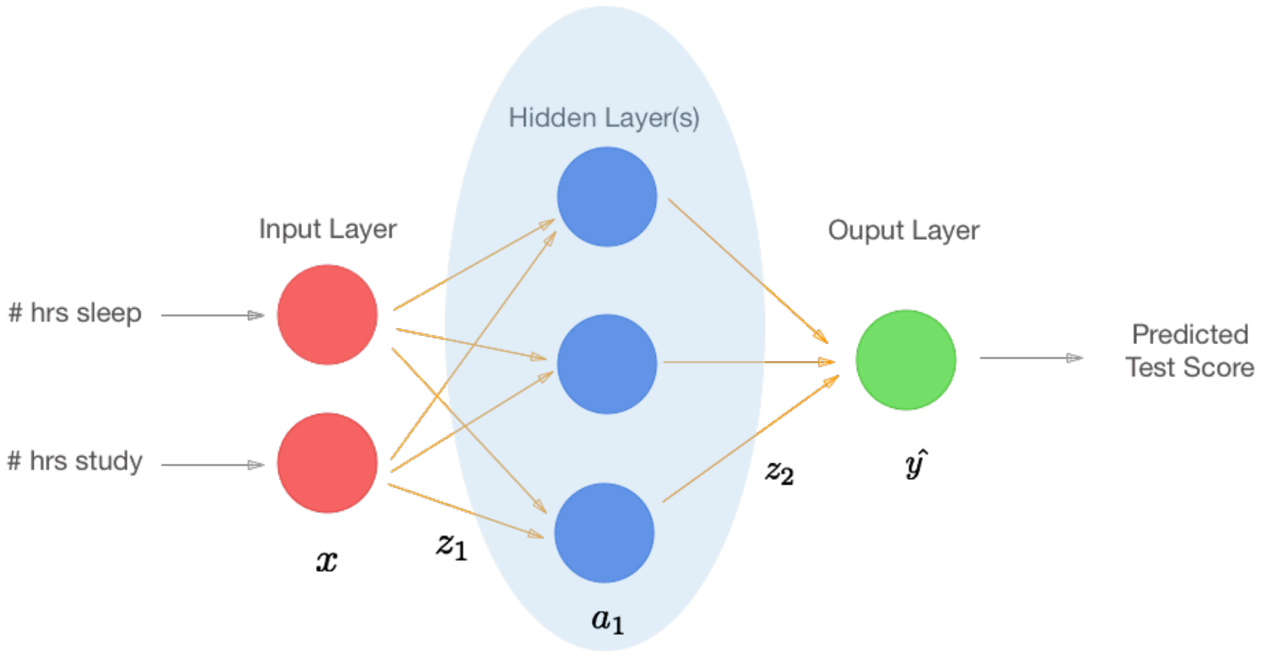


Diagram 1: An example of a neural network

1.2.1 Forward Propagation

The first step in a neural network is the forward propagation. Given an input, the network makes a prediction on what the output would be. To propagate the input across the layers, we perform functions like that of below:

$$
\begin{aligned}
&z_1 = xW_1 + b_1 \\
&a_1 = \tanh(z_1) \\
&z_2 = a_1W_2 + b_2 \\
&\hat{y} = softmax(z_2)
\end{aligned}
$$

The equations z_1, z_2 are linear functions with x as input and W, b are weights and biases. The a_1 in the hidden linear performs a nonlinear activation function $tanh$. The $tanh$ function takes in the inputs z_1 and output values in the range of [-1. 1]. In general, the activation function condenses very large or very small values into a logistic space, and their relatively simple derivatives allow for gradient descent to be workable in backpropagation. In the output layer, we perform the softmax function that turns the values of z_2 into a probability distribution where the highest value is the predicted output. The equations above are the steps that occur in the forward propagation. The next step is backpropagation, which is where the actual learning happens.

1.2.2 Backpropagation

Backpropagation is a way of computing gradients of expressions through recursive application of chain rule. [11] The backpropagation involves two steps: calculating the loss, and performing a gradient descent. We calculate the loss L by cross entropy loss to determine how off our predicted output $\hat{y}$ is from the correct output y. We typically think of the input x as given and fixed, and the weights and biases as the variables that we are able to modify. Because we randomly initialize the weights and biases, we expect our losses to be high at first. The goal of training is to adjust these parameters iteration by iteration so that eventually the loss is minimized as much as possible. We need to find the direction in which the weight-space improves the weight vector and minimizes our loss is the gradient descent.

The gradient descent is an optimization function that adjusts weights according to the error. The gradient is another word for slope. The slope describes the relationship between the network's error and a single weight, as in how much the error changes as the weight is adjusted. The relationship between network error and each of those weights is a derivative, $\frac{\delta L}{\delta W}$, which measures the degree to which a slight change in a weight causes a slight change in the error. [10] The weights are represented in matrix form in the network, and each weight matrix passes through activations and sums over several layers. Therefore, in order to find the derivative we need to use the chain rule to calculate the derivative of the error in relation to the weights. If we were to apply the backpropagation formula for the equations listed from the forward propagation section, we have the following derivatives for the weights in respect to the loss:

$$z_1 = xW_1 + b_1$$
$$a_1 = \tanh(z_1)$$
$$z_2 = a_1W_2 + b_2$$
$$\hat{y} = softmax(z_2)$$

$$\frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta a_1} \cdot \frac{\delta a_1}{\delta z_1} \cdot \frac{\delta z_1}{\delta W_1} = x^T \cdot (1 - tanh^2\, z_1) \cdot \hat{y} - y \cdot W_2^T$$

$$\frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta W_2} = a_1^T \cdot \hat{y} - y$$

We continually adjust the model's weights in response to the error it produces iteration after iteration until the error can no longer be reduced.

1.3 Recurrent Neural Networks

1.3.1 Forward Propagation

While traditional feedforward neural networks perform well in classification tasks, they are limited to looking at individual instances rather than analyzing sequences of inputs. Sequences can be of arbitrary length, have complex time dependencies and patterns, and have high dimensionality. Some examples of sequential datasets include text, genomes, music, speech, text, handwriting, change of price in stock markets, and even images, which can be decomposed into a series of patches and treated as a sequence. [10] Recurrent neural networks are built upon neurons like feedforward neural networks but have additional connections between layers.

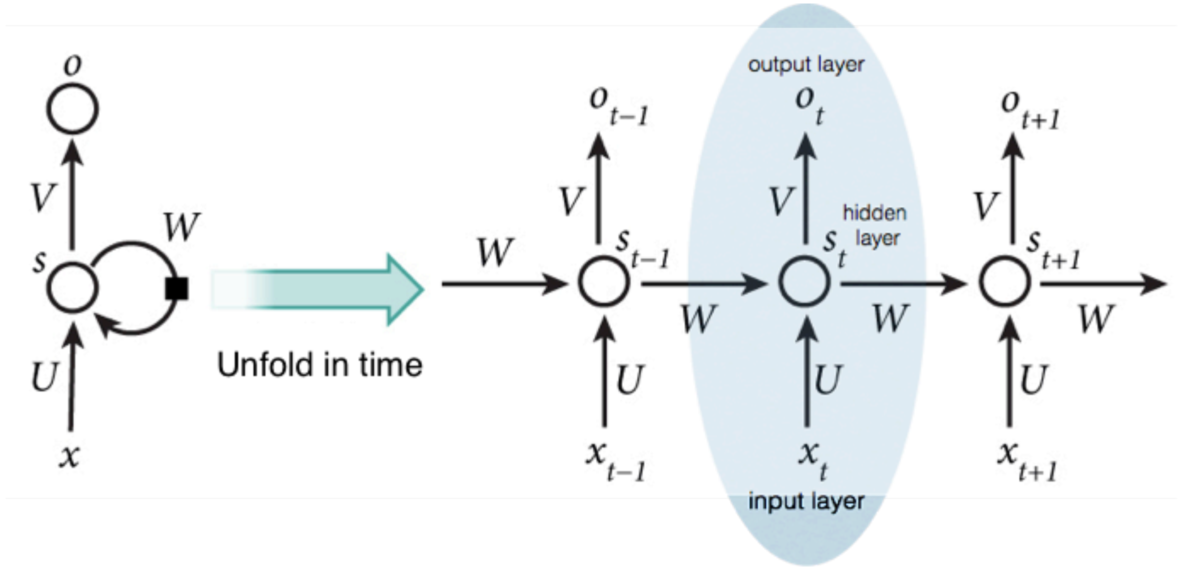


Diagram 2: Recurrent Neural Network in time steps [2]

The diagram above illustrates how the workings of the RNN, when unfolded in time, is very similar to feedforward neural networks. The area highlighted in blue is similar to what is happening in the diagram of the feedforward neural network. There is the input layer x_t, hidden layer s_t, and output layer o_t. $U, V,$ and $W$are the parameters or the weights that the model needs to learn. The difference between the feedforward neural network and the RNN is that there is an additional input, s_{t-1}, fed into the hidden layer s_t. If the network path highlighted in blue is the current time step t, then the previous that is the network at timestep *t-1*, and the network after happens at time step *t+1*, in which the current hidden layer s_t will be fed into s_{t+1} along with x_{t+1}. In the hidden layer, we apply an activation function to the sum of the previous hidden layer state and current input x_t (in the below diagram, the $tanh$ activation function is applied). While the left hand side of diagram 2 seems to suggest that RNNs have a cyclic cycle, the connection between previous time step and current time step in the hidden state is still acyclic; this is important to recognize because the network needs to be acyclic in order for backpropagation to be possible. The diagram below illustrates what is happening in the hidden state of a RNN:

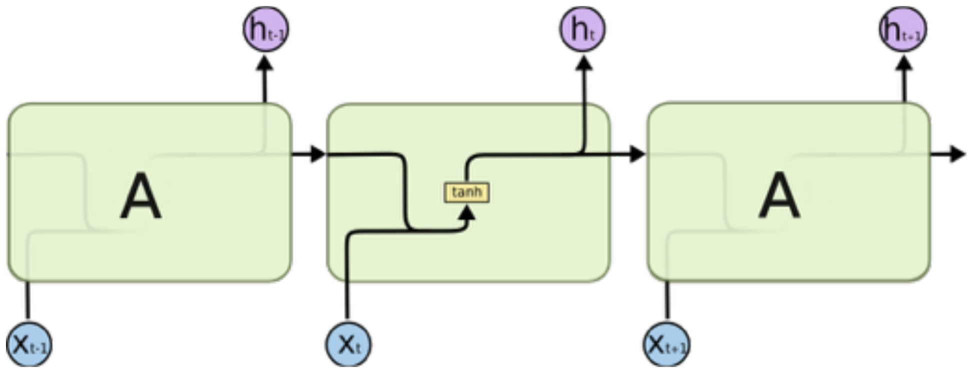


Diagram 3: Hidden State of a RNN [16]

Mathematically, we represent the step happening in the hidden state h_t as:

$$h_t = tanh(Wx_t + Uh_{t-1})$$

W and U are weight matrices; they are filters that determine how much importance to accord to both the present input and the previous hidden state. When we feed in the previous hidden state, it contains traces of all those that preceded h_{t-1}; this is how the RNN is able to have a persistent memory. [10]

1.3.2 Backpropagation Through Time

In the backpropagation, we calculate the error the weight matrices generate, and then adjust their weights until the error cannot go any lower. As we see in diagram 2, the weight matrix W is carried through each time step. In order to compute the gradient for the current W, we need to perform the chain rule through a series of previous time steps. Because of this, we call the process back propagation through time (BPTT). If the sequences are quite long, the BPTT can take a long time; thus, in practice many people truncate the backpropagation to few steps instead of all the way to the beginning. [10]

1.3.3 Issue of Vanishing Gradient

While in theory the RNN should retain memory through the time steps, in practice RNN performed poorly. Hochreiter (1991) and Bengio, et al. (1994) explored the problem in depth of why gradient based learning algorithms face an increasingly difficult problem as the duration of the dependencies to be captured increases. [8] One major issue is the vanishing gradient problem. As we mentioned before, the gradient is the derivative of the loss with respect to the weights. If the gradient is so small, we cannot adjust the weights in a direction that decreases the error, and so the network cannot learn. In a RNN, the layers and time steps of deep neural networks relate to each other through multiplication. Multiplying a number slightly greater than one can make the number become immeasurably large (exploding), and multiplying a number slightly less than one can diminish to zero very fast (vanishing). [10] Therefore, derivatives or gradients are susceptible to vanishing or exploding in a RNN. We can solve exploding gradients but truncating or squaring the values, but resolving vanishing gradients is harder. Below is a diagram of the graphs of the tanh function and its derivative.

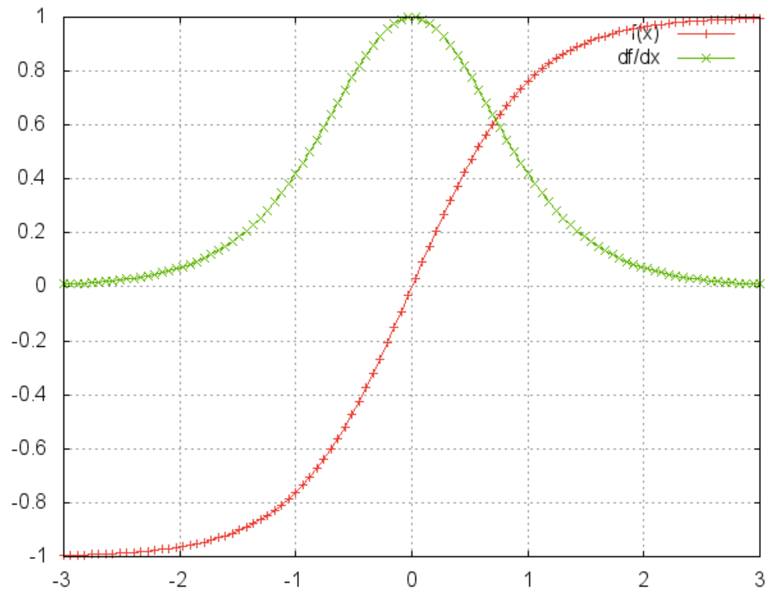


Diagram 4: Graphs of tanh and its derivative [1]

The tanh activation function maps output values in the range of [-1,1] and the maximum value of the derivative is 1 with 0 at both ends. Weight matrices are randomly initialized to be small values, and with a derivative that is slightly less than 1, multiplying the derivatives across the previous time steps can cause the gradients to vanish very rapidly. [15] This prevents learning long-term dependencies and is the cause for RNNs to perform poorly. While there are tricks to overcome this issue, it does not change the fact that RNNs fundamentally have unstable gradients that can vanish and explode quickly. [15] In the next section, we discuss Long Short-Term Memory Networks, a type of RNN that was discovered in the mid 1900s in order to overcome the issue of vanishing gradients.

[1] "Transfer Function Layers." *Nn*. Read the Docs, n.d. Web. 06 May 2016. <http://nn.readthedocs.io/en/rtd/transfer/#tanh>.

1.4 Long Short-Term Memory

1.4.1 The Cell State and the Three Gates

LSTM was first introduced in 1997 by Sepp Hochrieiter and Jürgen Schmidhuber. LSTMs are capable of bridging time intervals in excess of 1000 time steps even in case of noisy, incompressible input sequences, without loss of short time lag capabilities. [9] The architecture enforces constant error flow through internal states of special unit known as the memory cell.

There are three gates to the cell: the forget gate, input gate, and output gate. These gates are sigmoid functions that determine how much information to pass or block from the cell. Sigmoid functions takes in values and outputs them in the range of [0,1]. In terms of acting as a gate, a value of 0 means let nothing through, and a value of 1 means let everything through. These gates have their own weights that are adjusted via gradient descent. For the rest of the explanation for the forward propagation of the LSTM, we will refer to the diagram below.

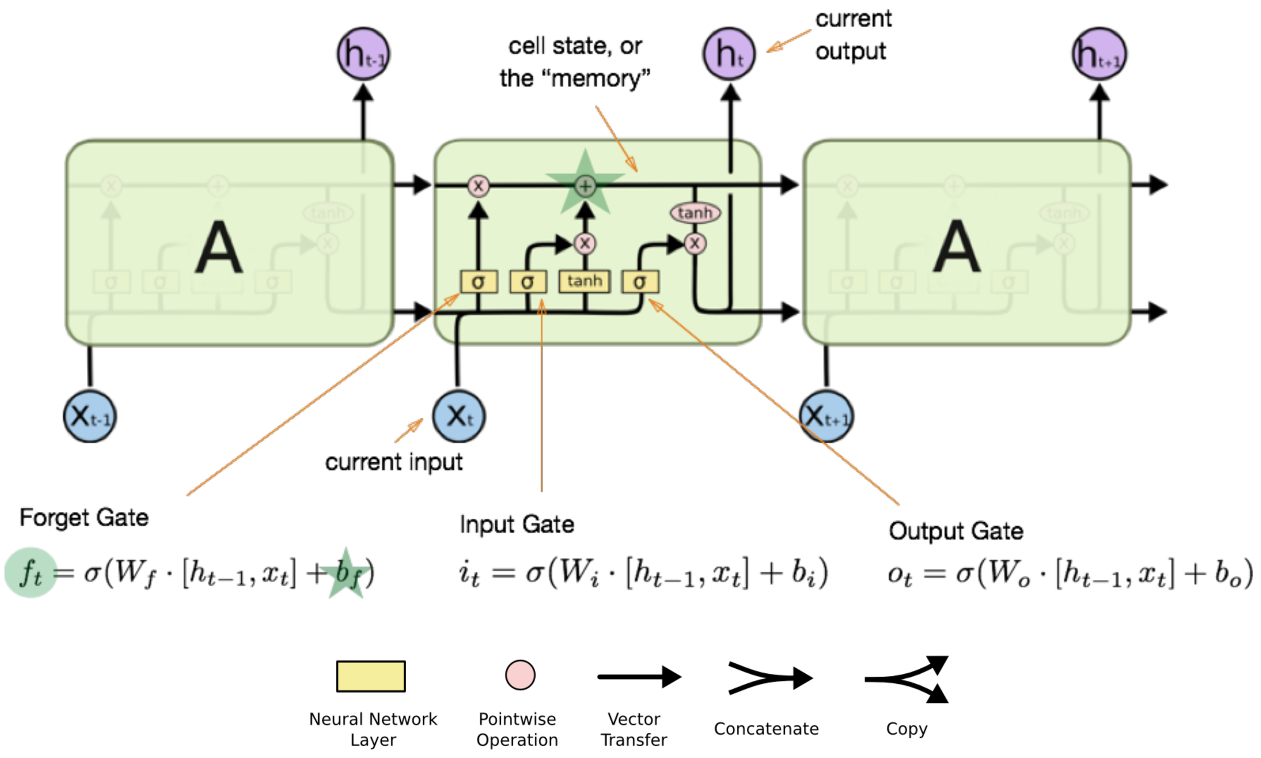


Diagram 5: The hidden state of a LSTM [17]

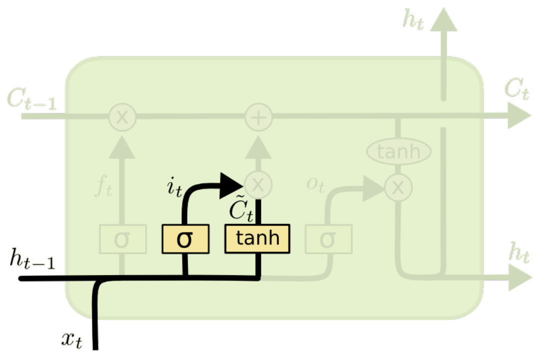
In the equations listed under the forget gate, input gate, and output gate in the diagram, h_{t-1} is the previous hidden state, x_t is the current input, W is the weight matrix, and b is the bias.

1.4.2 Forget Gate

The first step is the forget gate, in which the sigmoid function outputs a value ranging from 0 to 1 to determine how much information of the previous hidden state and current input it should retain. Forget gates are necessary to performance of LSTM because the network does not necessarily need to remember everything that has happened in the past. For example, if we are moving from one music piece to the next in the input dataset, then we can forget all of the information related to the old music piece.

1.4.3 Input Gate

The next step involves two parts. First, the input gate determines what new information to store in the memory cell. Next, a tanh layer creates a vector of new candidate values to be added to the state. From the example of the music learning model, we are inputting the first few sequences of notes of the new piece.

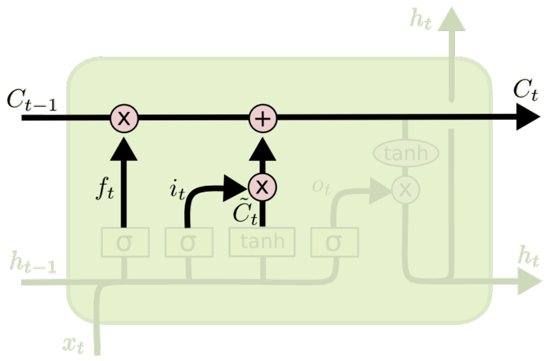


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Diagram 6: Input gate [17]

1.4.4 Updating The Cell Memory

At this point we have determined what to forget and what to input, but we have not actually changed the memory cell state.



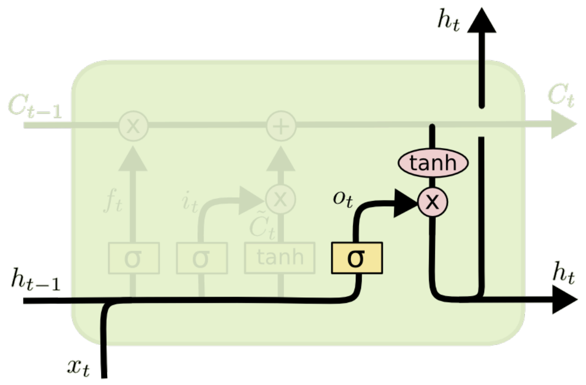
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Diagram 7: Updating the cell memory [17]

To update the old cell state, C_{t-1}, we multiply the vector f_t, and then add $i_t \ * \ \tilde{C}_t$.

1.4.5 Output gate

To determine what to output from the memory cell, we again apply the sigmoid function to the previous hidden state and current input, then multiply that with tanh applied to the new memory cell (this will make the values between -1 and 1). In the music learning model example, we want to output information that will be helpful in predicting the next sequence of notes. Perhaps information such as the time signature and key of the new music piece would be outputted.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Diagram 8: Output gate [17]

1.4.6 Why LSTM is superior over RNN

The extra complications with the gates may make it difficult to see why exactly the LSTM is better than the RNN. LSTM has an actual memory built into the architecture that lacks in RNN. We update the cell memory with new information ($i_t * \tilde{C}_t$) by addition, highlighted with a green star in the diagram 5, and that makes the LSTM maintain a constant error when it must be backpropagated at depth. [10] Instead of determining the subsequent cell state by multiplying its current state with the new input, the addition prevents the gradient from exploding or vanishing. [10] (Although we do still have to multiply the forget gate to the memory cell.)

1.5 Brief Overview of Training, Testing, and Validation

Before diving into the two implementations, we will cover some machine learning concepts in this section.

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. [7] The two LSTM implementations solve 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. [7] The difference in validation set and test set is that the validation set is used to tune the parameters of the network based on the error rate. From the validation we pick the model that performs the best. The test set is then strictly used to assess the performance of the chosen model, and therefore no tuning must happen during 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.

There can be cases when the error rate during training can be very low while the testing error rate is much higher. This phenomenon is called overfitting. The test error, also referred to as generalization error, is the expected error of the model on previously unseen records. [19] In training, we may be tempted to increase the complexity of the model to produce good results. However, sometimes increasing the complexity does not necessarily produce a model that generalizes well to test examples. A good model should have both a low training and generalization error. Model underfitting can also occur if the model has yet to learn the true structure of the data; usually in this case the training error and the generalization error will both be high. In the case of underfitting, the issues can be not enough dataset, or the model is not complex enough. While there are theories that address possible issues to these solutions, there is quite an art in training neural networks.

2. Implementation for Generating Traditional Chinese Music

2.1 Overview of Andrej Karpathy's char-RNN Model

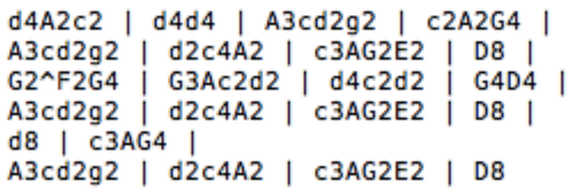
The char-RNN code written by Andrej Karpathy takes a single text file as an input and feeds it into the RNN algorithm that learns to predict the next character in the sequence. After training the RNN, it can generate text character by character that looks stylistically similar to the original dataset. The code is written in Lua and uses Torch. Useful features in this code are: option to have multiple layers, supporting code for model checkpointing, and using mini-batches to make the learning process efficient.

2.2 Input Dataset: ABC notation

The Chinese music dataset I used as input for the char-RNN model is a collection of simple Chinese music tunes I found online. It is a combination of 2,000 songs from abcnotation.com (1,200 songs, webscraped) and from a Japanese online blog[2], also about 1,000 songs. The ABC notation was developed by Chris Walshow so that music can be represented using ASCII symbols. [21] 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: Chocolate Pudding
C: Ayako Mikami
M:6/8
K:D

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 meter, and K is the key. The notes portion looks something like below:



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

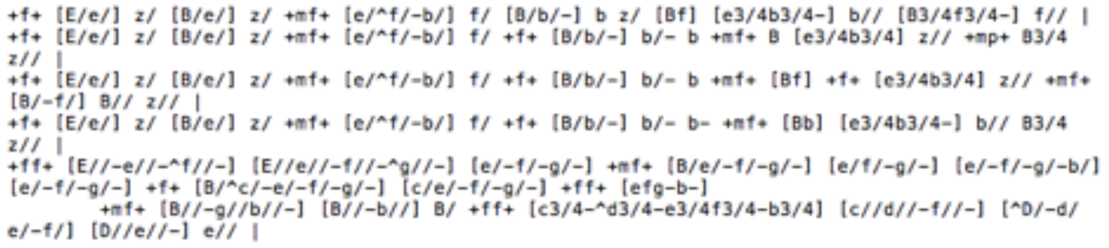
"D" stands for the note D, "C" for C, and so on.

[2] I could not locate the url of the website, which seemed like it was personally owned by a music hobbiest. There is a chance that the site has been taken down.

As Walshow explains,

> Upper case (capital) letters, CDEFGAB, are used to denote the bottom octave (C represents middle C, on the first leger line below the treble stave), continuing with lower case letters for the top octave, cdefgab (b is the one above the first leger line above the stave).To go down an octave, just put a comma after the letter and to go up an octave use an apostrophe. [21]

There are many other symbols and formatting involved in abc notation. The abc formatted songs are converted to MIDI (Musical Instrument Digital Interface) files. MIDI files are series of messages such as "note on", "note off", "note/pitch", "pitchbend", and many more. [1] These MIDI files are then finally converted to mp3 files. Chinese folk songs are typically monophonic, so the notations are straightforward with representation of the notes and the rhythm. When attempting to increase the size of the dataset by attempting to convert more complex music (non folk traditional songs) from MIDI to abc notation, the resulting abc notations were much more long and complicated. Below is a screenshot of part of a more complex traditional music in abc notation:



```
+f+ [E/e/] z/ [B/e/] z/ +mf+ [e/^f/-b/] f/ [B/b/-] b z/ [Bf] [e3/4b3/4-] b// [B3/4f3/4-] f// |
+f+ [E/e/] z/ [B/e/] z/ +mf+ [e/^f/-b/] f/ +f+ [B/b/-] b/- b +mf+ B [e3/4b3/4] z// +mp+ B3/4
z// |
+f+ [E/e/] z/ [B/e/] z/ +mf+ [e/^f/-b/] f/ +f+ [B/b/-] b/- b +mf+ [Bf] +f+ [e3/4b3/4] z// +mf+
[B/-f/] B// z// |
+f+ [E/e/] z/ [B/e/] z/ +mf+ [e/^f/-b/] f/ +f+ [B/b/-] b/- b- +mf+ [Bb] [e3/4b3/4-] b// B3/4
z// |
+ff+ [E//-e//-^f//-] [E//e//-f//-^g//-] [e/-f/-g/-] +mf+ [B/e/-f/-g/-] [e/f/-g/-] [e/-f/-g/-b/]
[e/-f/-g/-] +f+ [B/^c/-e/-f/-g/-] [c/e/-f/-g/-] +ff+ [efg-b-]
        +mf+ [B//-g//b//-] [B//-b//] B/ +ff+ [c3/4-^d3/4-e3/4f3/4-b3/4] [c//d//-f//-] [^D/-d/
e/-f/] [D//e//-] e// |
```

Mixing the simple and complex abc notations together gave poor results and the network was not able to properly learn the abc notation.

2.3 Results

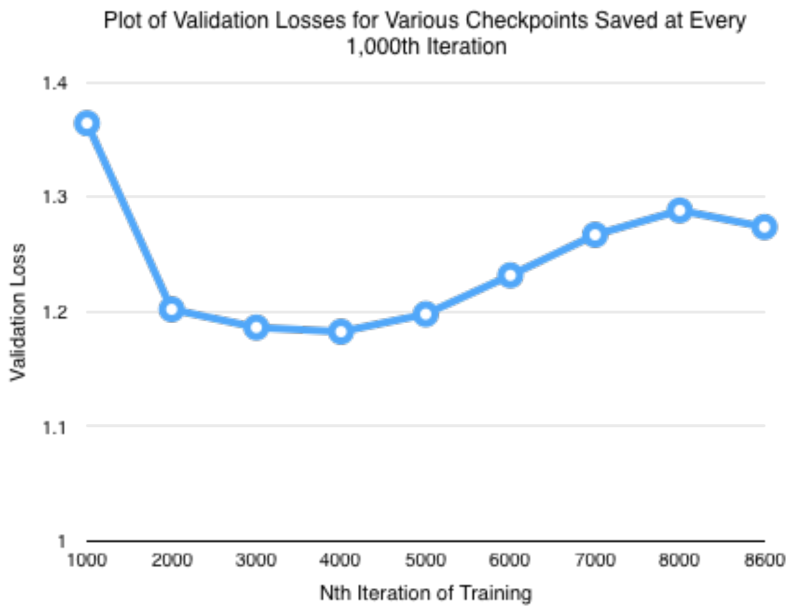
2.3.1 Checkpoints and Minimum Validation Loss

During training, a checkpoint is saved every 1,000 iterations and at every checkpoint, a filename that looks something like this is printed to the terminal:

```
lm_lstm_epoch0.95_2.0681.t7
```

The checkpoint file saves the current values for all the weights in the model. The number after epoch0.95 (which means it has almost complete one full pass through the training data) indicates the loss on validation data, which is 2.0681. The smaller the loss, the better the checkpoint file works when generating the music. Due to possible overfitting, the minimum validation loss is not necessarily at the end of the training. For example, the table and plot below show all the saved checkpoints during a single training:

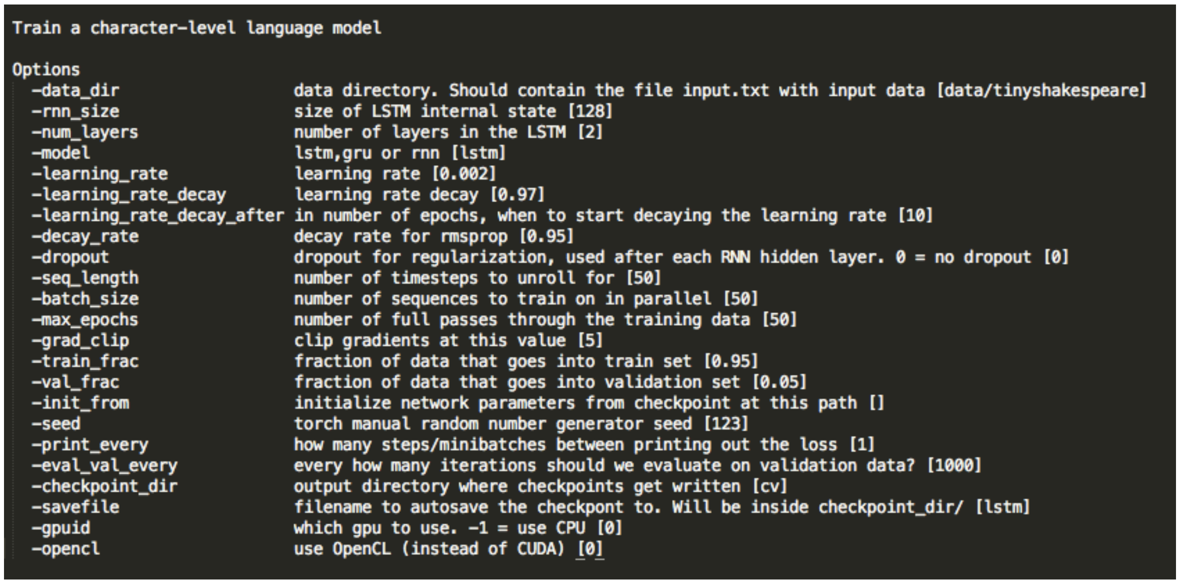
Nth Iteration (out of 8600)	Validation Loss
1000	1.3641
2000	1.2020
3000	1.1862
4000	**1.1825**
5000	1.1978
6000	1.2316
7000	1.2669
8000	1.2882
8600	1.2737



The minimum validation loss occurs at the 4,000th iteration out of 8,600 iterations. Rerunning the code on the same dataset produced the same loss values. Each iteration on average takes about 0.33 seconds, and in total the model takes about 2-3 hours to train on CPU.

2.3.2 Decreasing the batch parameter

The two files needed to run the char-rnn code are train.lua and sample.lua. The train.lua gives the several options to adjust the parameters to create the best model for a given dataset. The parameters with the default values in brackets are as below:

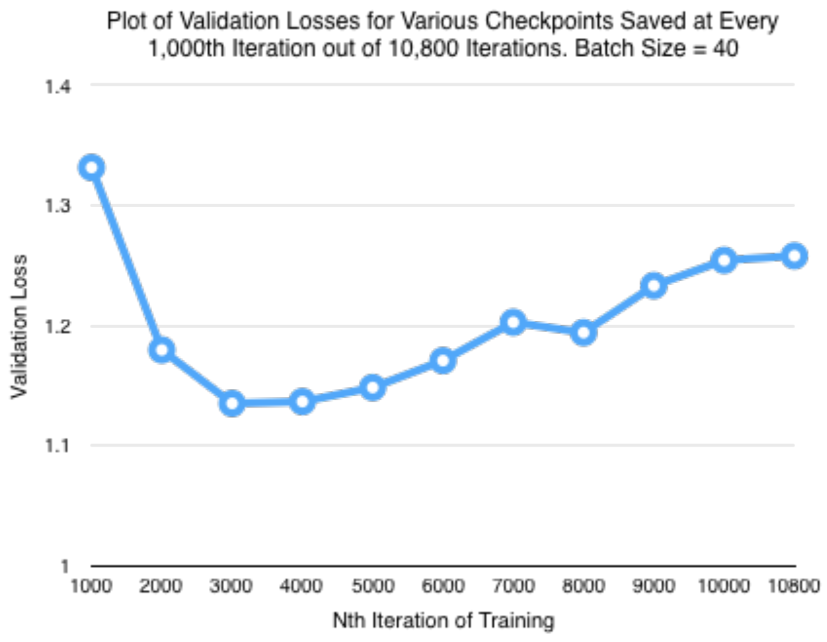


```
Train a character-level language model

Options
  -data_dir                  data directory. Should contain the file input.txt with input data [data/tinyshakespeare]
  -rnn_size                  size of LSTM internal state [128]
  -num_layers                number of layers in the LSTM [2]
  -model                     lstm,gru or rnn [lstm]
  -learning_rate             learning rate [0.002]
  -learning_rate_decay       learning rate decay [0.97]
  -learning_rate_decay_after in number of epochs, when to start decaying the learning rate [10]
  -decay_rate                decay rate for rmsprop [0.95]
  -dropout                   dropout for regularization, used after each RNN hidden layer. 0 = no dropout [0]
  -seq_length                number of timesteps to unroll for [50]
  -batch_size                number of sequences to train on in parallel [50]
  -max_epochs                number of full passes through the training data [50]
  -grad_clip                 clip gradients at this value [5]
  -train_frac                fraction of data that goes into train set [0.95]
  -val_frac                  fraction of data that goes into validation set [0.05]
  -init_from                 initialize network parameters from checkpoint at this path []
  -seed                      torch manual random number generator seed [123]
  -print_every               how many steps/minibatches between printing out the loss [1]
  -eval_val_every            every how many iterations should we evaluate on validation data? [1000]
  -checkpoint_dir            output directory where checkpoints get written [cv]
  -savefile                  filename to autosave the checkpont to. Will be inside checkpoint_dir/ [lstm]
  -gpuid                     which gpu to use. -1 = use CPU [0]
  -opencl                    use OpenCL (instead of CUDA) [0]
```

These default parameters worked well for the dataset. As an experiment, I decreased the batch size from the default size of 50 to 40. 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] x [sequence length] (length of each step, which is also the length at which the gradients can propagate backwards in time). [14] These chunks get allocated across two splits: training, and testing. By default the training fraction size (*train_frac*) is 0.95 and the validation fraction size (*val_frac*) is 0.05, meaning 95% of the data gets trained and 5% is used to estimate validation loss. With a small dataset, there could be very few chunks in total (~100 is considered small according to Karpathy). With the initial parameter settings for batch size and sequence length, the chunks were 172 training and 10 for validation, in total 182 chunks. By decreasing the batch size to 40, there were now 216 chunks for training and 12 for validating, so 228 in total.

The resulting validation loss was less than when trained with batch size of 50, as shown in the table and plot below.

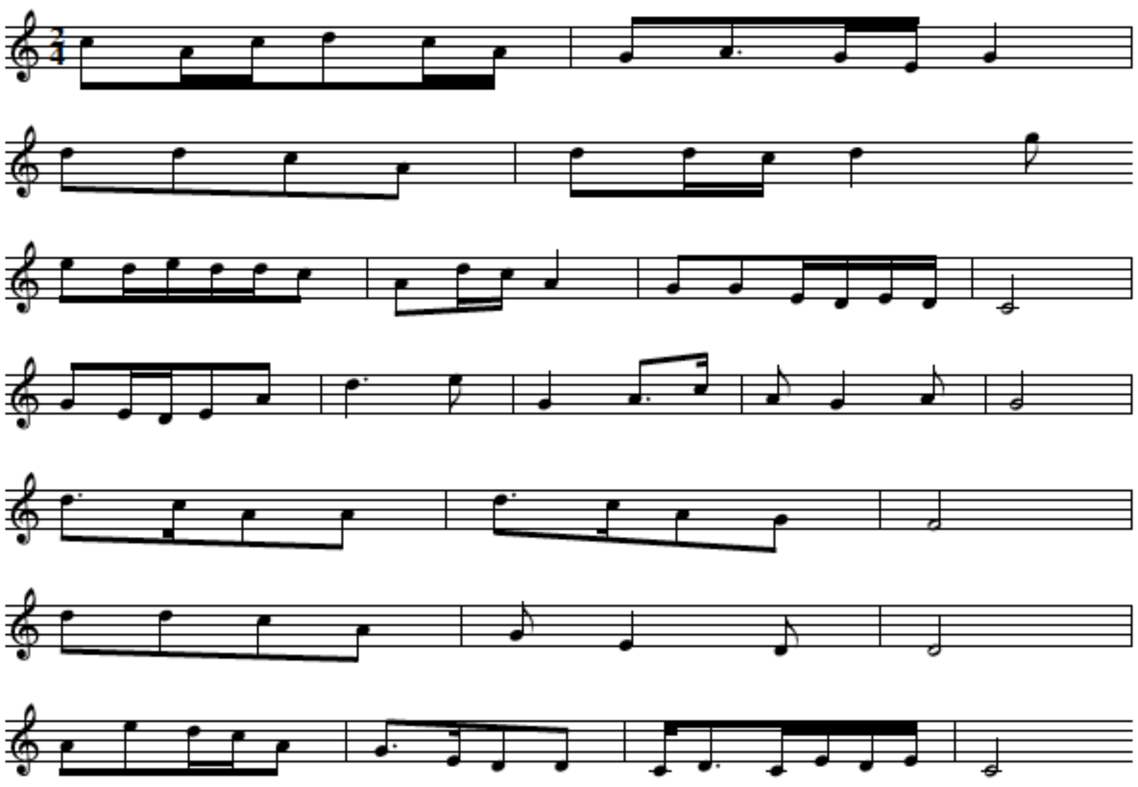
Nth Iteration (out of 8600)	Validation Loss
1000	1.3316
2000	1.1798
3000	1.1353
4000	1.1368
5000	1.1485
6000	1.1709
7000	1.2028
8000	1.1942
9000	1.2333
10000	1.2544
10800	1.2581



The minimum validation loss is 1.1353 at 3,000th iteration.

Below is a sample output text from the model (default parameters) and a musical score sheet of the output.

```
X:379
T: Xing kao zidigeng
N: C0233
O: China,Ã¿Anhui, Noliben
S: III, 361]
R: Wuge, Xiuqinzu. Maiku]
M: 2/4
L: 1/16
K: C
c2Acd2cA | G2A3GEG4 |
d2d2c2A2 | d2dcd4g2
e2deddc2 | A2dcA4 | G2G2EDED | C8 |
G2EDE2A2 | d6e2 | G4A3c | A2G4A2 | G8 |
d3cA2A2 | d3cA2G2 | F8 |
d2d2c2A2 | G2E4D2 | D8 |
A2e2dcA2 | G3ED2D2 | CD3CEDE | C8
```



2.4 Challenges and Performance Evaluation

The challenge of getting a successful result from this char-RNN was the limitation of the small dataset. There are not many Chinese music 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 I ran the char-RNN it was on a dataset of 277KB, which is far less than the minimum required size of 1MB for the char-RNN to produce tangible results. Evidently, the result was poor. After converting the output to a mp3 file and listening to the song, there were about only 2 parts (~1-2 seconds) that sounded "Chinese" in the 34 seconds and rest were random sequences of notes that made no musical "sense." After finding more music from another source, the dataset increased to 457KB, and the results were significantly better. Overall, music generated from the model with the larger dataset stylistically sounded Chinese. 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. Since I have very little background in Chinese Music Theory, my evaluation is subjective and dependent on how the music sounds like. Regardless of the inability to quantitatively measure the accuracy of how well the model learned Chinese music, I am still impressed by LSTM's ability to generate music that at least sounds very similar to the given input dataset.

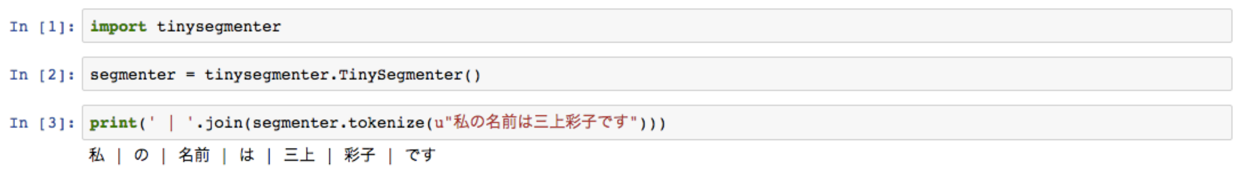
3. Implementation for Generating Japanese Lyrics

To better understand LSTMs, I wanted to work on my own implementation. Because this was my first time actually trying to code a LSTM model, I wanted to work with a easier dataset than music, and thus I chose to implement a LSTM model that generates Japanese lyrics. For this second implementation, I modified Denny Britz's GRU model into an LSTM model. [1] GRU is another type of RNN model that has a different architecture from LSTM. The implementation is written in Python and uses the Theano library. For faster runtime I ran the code on GPU, specifically Amazon Web Services EC2 g2.2xlarge instance.

3.1 Collecting and Preprocessing the Data

The program will read in a single text file that contains all the Japanese lyrics. Since there are no online sources that provide Japanese lyrics that we can copy and paste (due to legal copyright issues) I used an application called Lyrics Master to lookup the lyrics and copy and paste them into a text file.

The next step is to preprocess the dataset. Unlike the char-RNN model that parses the dataset char by char, this one parses word by word. Since we want to make the model specific to learning Japanese, we use a Japanese tokenizer written in Python by Masato Hagiwara called TinySegementer. Below is an example of how the TinySegmenter parses the text "My name is Ayako Mikami."



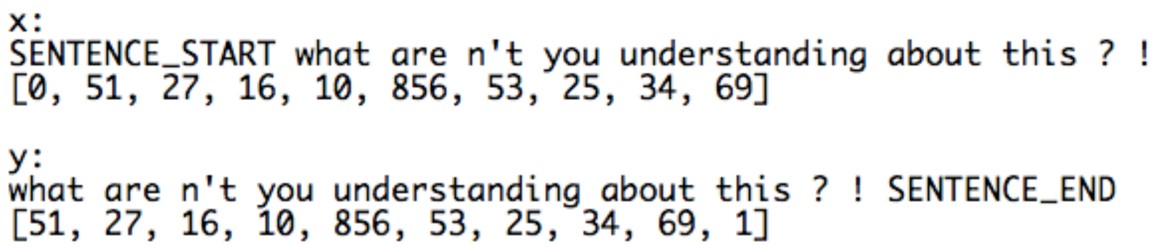
```
In [1]: import tinysegmenter

In [2]: segmenter = tinysegmenter.TinySegmenter()

In [3]: print(' | '.join(segmenter.tokenize(u"私の名前は三上彩子です")))
        私 | の | 名前 | は | 三上 | 彩子 | です
```

After parsing the lyrics into words, we build a vocabulary list. The rational for a vocabulary list is that many words will only appear once or twice in the lyrics text file. The model will not learn these words properly anyway because of their low frequency. We will set the vocabulary size to 8000. The top 8000 frequent words in the text file will be in the vocabulary size. If the model encounters a word not in the vocabulary list, we replace the word with UNKNOWN_TOKEN, and we will predict it like any other word. To each lyric in the lyrics text file we append START_TOKEN and END_TOKEN. When we actually generate a new lyric from the learned model, we can use the START_TOKEN to feed into the model, and from there the model can predict the word for the beginning of the lyric.

The neural network takes in vectors as inputs, not strings. To convert the strings to vectors we create a mapping between words and indices – index_to_word and word_to_index. Britz explains how the input and corresponding labels are set up:

> For example, the word "friendly" may be at index 2001. A training example x may look like [0, 179, 341, 416], where 0 corresponds to SENTENCE_START. The corresponding label y would be [179, 341, 416, 1]. Remember that our goal is to predict the next word, so y is just the x vector shifted by one position with the last element being the SENTENCE_END token. In other words, the correct prediction for word 179 above would be 341, the actual next word. [2]

Below is a single training example, where x is the input and y is the correct prediction. What we are actually feeding in are the vectors:

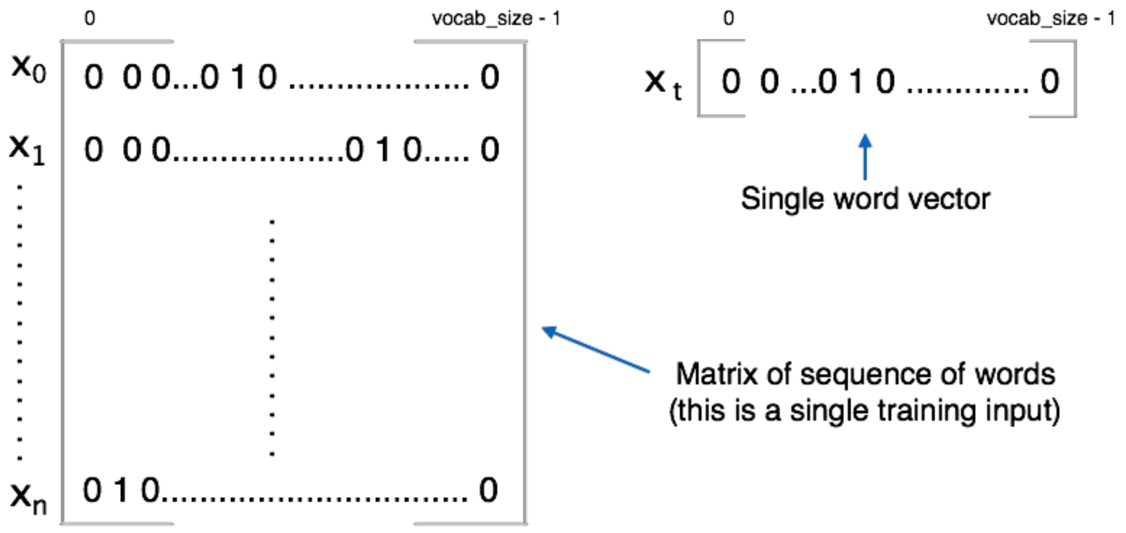


```
x:
SENTENCE_START what are n't you understanding about this ? !
[0, 51, 27, 16, 10, 856, 53, 25, 34, 69]

y:
what are n't you understanding about this ? ! SENTENCE_END
[51, 27, 16, 10, 856, 53, 25, 34, 69, 1]
```

Figure 1: Sample input x and its corresponding labels y [3]

I wrote my own preprocessing code so to parse and encode Japanese text, but the mapping idea remains the same. Please refer to the appendix for the code.

The input x is a sequence of words, and each x_t is a single word. However, converting the strings into corresponding numerical indices is not enough. In order to make the matrix multiplication work, we cannot use the word index as input. Instead, we need to represent each word as a one-hot vector of size 8000, our *vocabulary_size*. For example, the word with index 10 will be vector of all 0's and a 1 at position 10. Each word, x_t becomes a vector, and x is a matrix, with each row representing a word. The below figure illustrates what the input matrix looks like:



Given that the vocabulary size is 8000 and we have a hidden layer size of 100, then the dimensions of the matrices and vectors for the LSTM network will look like this:

$$
\begin{aligned}
x_t &\in \mathbb{R}^{8000} \\
o_t &\in \mathbb{R}^{8000} \\
s_t &\in \mathbb{R}^{100} \\
c_t &\in \mathbb{R}^{100} \\
U &\in \mathbb{R}^{100 \times 8000} \\
V &\in \mathbb{R}^{8000 \times 100} \\
W &\in \mathbb{R}^{100 \times 100}
\end{aligned}
$$

x_t is a word vector input, o_t is an output vector, s_t is hidden layer vector, c_t is the cell state, and U, V, W are weight matrices that correspond to input layer, output layer, and hidden layer, respectively as illustrated in diagram 2**.**

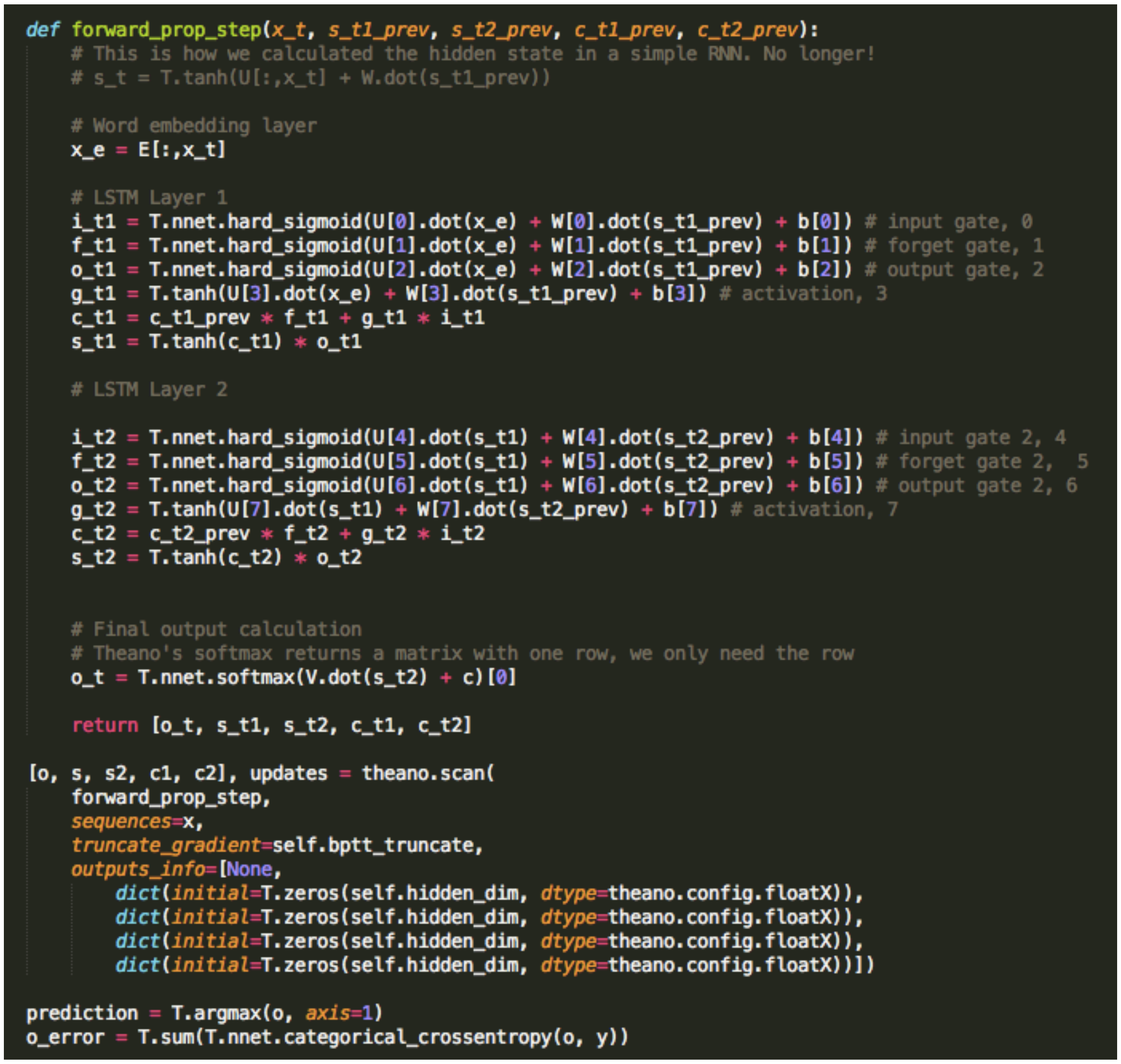
Knowing the dimensions of our matrices is important because it not only helps in making sure we set the correct initialization values, but also tells us the complexity of our model. As Britz explains:

> Remember that $U, V, and\ W$ are the parameters of our network we want to learn from data. Thus, we need to learn a total of $2HC + H^2$ parameters. In the case of $C = 8000$ and $H = 100$ that's 1,610,000. The dimensions also tell us the bottleneck of our model. Note that because x_t is a one-hot vector, multiplying it with U is essentially the same as selecting a column of U, so we don't need to perform the full multiplication. Then, the biggest matrix multiplication in our network is Vs_t. That's why we want to keep our vocabulary size small if possible. [2]

What follows after for implementation steps are: initialization of variables; forward propagation; calculating the loss; back propagation through time; gradient checking; stochastic gradient descent; and finally generating text. Please refer to Britz's blog post for further explanations.

3.2 Forward Propagation

As mentioned before, I modified Britz's implementation of GRU to make it LSTM. The biggest modification happens in the forward propagation, which looks like below:



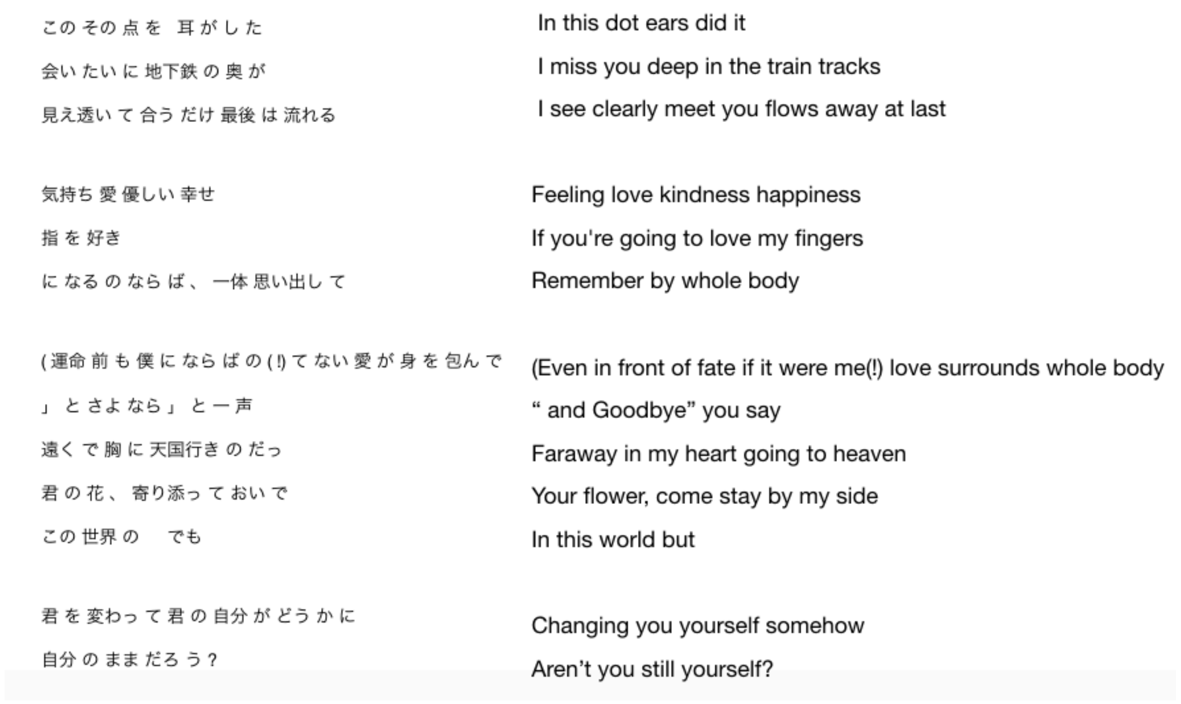
```
def forward_prop_step(x_t, s_t1_prev, s_t2_prev, c_t1_prev, c_t2_prev):
    # This is how we calculated the hidden state in a simple RNN. No longer!
    # s_t = T.tanh(U[:,x_t] + W.dot(s_t1_prev))

    # Word embedding layer
    x_e = E[:,x_t]

    # LSTM Layer 1
    i_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0]) # input gate, 0
    f_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1]) # forget gate, 1
    o_t1 = T.nnet.hard_sigmoid(U[2].dot(x_e) + W[2].dot(s_t1_prev) + b[2]) # output gate, 2
    g_t1 = T.tanh(U[3].dot(x_e) + W[3].dot(s_t1_prev) + b[3]) # activation, 3
    c_t1 = c_t1_prev * f_t1 + g_t1 * i_t1
    s_t1 = T.tanh(c_t1) * o_t1

    # LSTM Layer 2

    i_t2 = T.nnet.hard_sigmoid(U[4].dot(s_t1) + W[4].dot(s_t2_prev) + b[4]) # input gate 2, 4
    f_t2 = T.nnet.hard_sigmoid(U[5].dot(s_t1) + W[5].dot(s_t2_prev) + b[5]) # forget gate 2,  5
    o_t2 = T.nnet.hard_sigmoid(U[6].dot(s_t1) + W[6].dot(s_t2_prev) + b[6]) # output gate 2, 6
    g_t2 = T.tanh(U[7].dot(s_t1) + W[7].dot(s_t2_prev) + b[7]) # activation, 7
    c_t2 = c_t2_prev * f_t2 + g_t2 * i_t2
    s_t2 = T.tanh(c_t2) * o_t2


    # Final output calculation
    # Theano's softmax returns a matrix with one row, we only need the row
    o_t = T.nnet.softmax(V.dot(s_t2) + c)[0]

    return [o_t, s_t1, s_t2, c_t1, c_t2]

[o, s, s2, c1, c2], updates = theano.scan(
    forward_prop_step,
    sequences=x,
    truncate_gradient=self.bptt_truncate,
    outputs_info=[None,
        dict(initial=T.zeros(self.hidden_dim, dtype=theano.config.floatX)),
        dict(initial=T.zeros(self.hidden_dim, dtype=theano.config.floatX)),
        dict(initial=T.zeros(self.hidden_dim, dtype=theano.config.floatX)),
        dict(initial=T.zeros(self.hidden_dim, dtype=theano.config.floatX))])

prediction = T.argmax(o, axis=1)
o_error = T.sum(T.nnet.categorical_crossentropy(o, y))
```

The model architecture has two hidden layers (commented above in the code as #LSTM Layer 1, #LSTM Layer 2). The activation function is tanh. I took the equations from diagram 5 and wrote it in Python using Theano's built-in functions.

3.3 Results

For this implementation, I needed to run the code on GPU, so I used Amazon Web Services EC2 g2.2xlarge. The dataset is about 1MB, or 732 Japanese lyrics. A single SGD step time takes on average 44 milliseconds, and the entire model takes roughly 3 hours to train.

Here is a sample training output, with the English translation to the right:



この その 点 を　耳 が し た	In this dot ears did it
会い たい に 地下鉄 の 奥 が	I miss you deep in the train tracks
見え透い て 合う だけ 最後 は 流れる	I see clearly meet you flows away at last
気持ち 愛 優しい 幸せ	Feeling love kindness happiness
指 を 好き	If you're going to love my fingers
に なる の なら ば 、 一体 思い出し て	Remember by whole body
(運命 前 も 僕 に なら ば の (!) て ない 愛 が 身 を 包ん で	(Even in front of fate if it were me(!) love surrounds whole body
」 と さよ なら 」 と 一 声	“ and Goodbye” you say
遠く で 胸 に 天国行き の だっ	Faraway in my heart going to heaven
君 の 花 、 寄り添っ て おい で	Your flower, come stay by my side
この 世界 の　　でも	In this world but
君 を 変わっ て 君 の 自分 が どう か に	Changing you yourself somehow
自分 の まま だろ う ?	Aren’t you still yourself?

The result is quite poor. There are multiple areas where the grammar is incorrect, and the sentences do not make any sense. It did learn that the lyric composition is 3-4 lines of text, a line break, another 3-4 lines of text. The poor performance could either be due to a small dataset, not enough optimization in the code, error in the code, or combination of any of them. In addition, removing any English words in the preprocessing step would be ideal, since including both Japanese and English complicates the learning.

At this point I could not determine what the exact reason for, I decided to increase the dataset. To increase the dataset, instead of batching a single input x as the entire lyrics piece, I made the single input x to be a single line of lyrics. This makes it so that from 732 inputs to around 31,000 inputs. I also collected 2MB of literature written by Souseki Natsume. After running the code over the larger dataset, there seems to be a bug in the code as there were error messages that halted the training.

4. Conclusions and Future Work

I would like to continue working on the implementation to fix the code and improve the performance of the model. There is still much to understand about LSTM networks both in theory and in practice. There are many optimization techniques involved in neural networks that I would like to further explore.

Overall I am impressed with the progress and results deep learning techniques have accomplished in the recent years. I am fascinated by deep neural networks' capability to create something that is in align with the style of the given input, and yet have its own unique taste that can be sometimes bizarre but sometimes incredible. I look forward to future human and machine collaborations in the fields of music, literature, and art.

Appendix

Please visit https://github.com/mikamia for all code related to this thesis.

References

[1] Amandaghassaei. "What Is MIDI?" Web log post. Instructables.com. Autodesk, Inc., 2012. Web. 18 Dec. 2015. <http://www.instructables.com/id/What-is-MIDI/>. Web. Page 1.

[2] Britz, Denny. "Language Model GRU with Python and Theano." *GitHub*. GitHub, 10 Nov. 2015. Web. 04 May 2016.

[3] Britz, Denny. "Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs." *WildML*. Wordpress, 17 Sept. 2015. Web. 04 May 2016.

[4] Britz, Denny. "Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients." *WildML*. Wordpress, 08 Oct. 2015. Web. 04 May 2016.

[5] Buduma, Nikhil. "Nikhil Buduma | A Deep Dive into Recurrent Neural Nets." *The Musings of Nikhil Buduma*. Nikhil Buduma, 11 Jan. 2015. Web. 04 May 2016.

[6] Buduma, Nikhil. "Nikhil Buduma | Deep Learning in a Nutshell." *The Musings of Nikhil Buduma*. Nikhil Buduma, 29 Dec. 2014. Web. 04 May 2016. <http://nikhilbuduma.com/2014/12/29/deep-learning-in-a-nutshell/>.

[7]Graves, Alex. *Supervised Sequence Labelling with Recurrent Neural Networks*. Vol. 385. N.p.: Springer, 2012. Print.

[8] Hastie, Trevor, Robert Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction: With 200 Full-color Illustrations*. 2nd ed. New York: Springer, 2001. Print.

[9] Hochreiter, S., Y. Bengio, P. Frasconi, and J. Schmidhuber. "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-term Dependencies." (2001): n. pag. *IEEE Press*. Web. 4 May 2016.

[10] Hochreiter, Sepp, and Jurgen Schmidhuber. "Long Short-Term Memory." *Neural Computation* (1997): n. pag. Web. 4 May 2016.

[11] "Introduction to Deep Neural Networks." *DL4J*. Ed. Deeplearning4j Development Team. Deeplearning4j: Open-source Distributed Deep Learning for the JVM, Apache Software Foundation License 2.0., 2016. Web. 4 May 2016.

[12] Karpathy, Andrej, Justin Johnson, and Li Fei-Fei. *Visualizing and Understanding Recurrent Networks* (2015): n. pag. *Cornell University Library*. Web. 4 May 2016.

[13] Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." *The Unreasonable Effectiveness of Recurrent Neural Networks*. Github, 21 May 2015. Web. 04 May 2016.

[14] Karpathy, Andrej. "Char-rnn." *GitHub*. GitHub, 21 May 2015. Web. 18 Dec. 2015.

[15] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning." *Nature Deep Review* 521 (2015): n. pag. Print.

[16] Nielson, Michael A. "CHAPTER 5." *Neural Networks and Deep Learning*. Determination Press, 2015. Web. 06 May 2016. <http://neuralnetworksanddeeplearning.com/chap5.html>.

[17] Olah, Christopher. "Calculus on Computational Graphs: Backpropagation." *Colah's Blog*. Github, 31 Aug. 2015. Web. 04 May 2016.

[18] Olah, Christopher. "Understanding LSTM Networks." *Colah's Blog*. Github, 27 Aug. 2015. Web. 04 May 2016. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

[19] Schmidhuber, J., F. Gers, and D. Eck. *Learning Nonregular Languages: A Comparison of Simple Recurrent Networks and LSTM*. Istituto Dalle Molle Di Studi Sull'Intelligenza Artificiale, 6 Jan. 2003. Web. 4 May 2016.

[20] Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005. Print.

[21] Walshaw, Chris. "How to Understand Abc (the Basics)." Web log post. Abc Notation Blog. WordPress, 23 Dec. 2009. Web. 18 Dec. 2015.

[22] Welch, Stephen C. "Neural Networks Demystified, Part 4: Backpropagation." Welch Labs. Welch Labs, 5 Dec. 2014. Web. 18 Dec. 2015.