# main

April 25, 2018

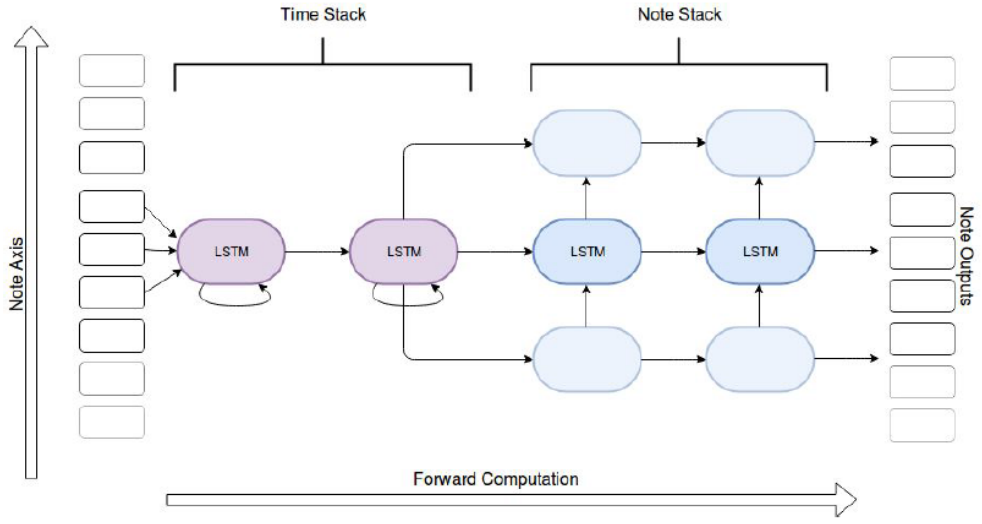
## 1 RNN Composing Music

### 1.0.1 Summary

In this project, we aimed to use Long Short Term Memory networks (LSTM) to generate classical and pop music. Here, we treated music as an object across two dimensions (time and note) and recreated a network structure capable of modeling a translationally invariant probability distribution across both time and note axis, conditioned upon information before. After trainning on 174 pieces of classical musics and 153 pieces of pop musics seperately for 50000 iterations, this model is able to produce polyphonic music with chords and noticeable musical structure. Finally, we generated hundreds of pieces of music and selected the most melodious ones, which are attached in the conclusion. We referred blog post "Composing Music with Biaxial RNNs" by Daniel Johnson in this project.

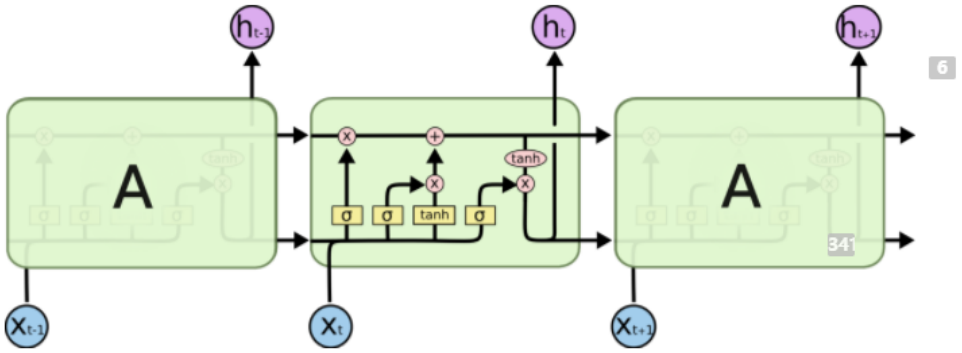
### 1.0.2 Introduction

In this project, we used biaxial architecture consisting of four layers, which are divided into two stacks, responsibling for time and note.




The inputs into each stack are rearranged such that the sequential axis is strictly along time or pitch, allowing the LSTM cell group to learn relative structure and conditional distributions along that dimension.

The LSTM is a special kind of RNN, which are capable of learning long-term dependencies. Because it can avoid long-term dependency probem, we used it in our model. The LSTM has a chain- structure and there are four neural network layer, interacting with each other in hidden layer.




The most important part for LSTM is that this model can add and remove information to the cell state by gates and sigmoid and tanh neural net layers.

Firstly, when the input message $h_{t-1}$ and $x_t$ walk through the sigmoid layer, which eliminates information that should be forgotten or gated. Here are the forget gate and the input gate:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_t)$$

Then, the messages walk through a tanh layer and add new information into our cell state:

$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C)$$

So the gated new input information will merge the old information which is deleted some information,

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

Finally, we decide the output $h_t$ by tanh and sigmod output gate:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

This is the basic idea of LSTM which was used to build our biaxial architecture. And we will metioned the model in the following parts.

### 1.0.3 Problem Formation and Design

We model the joint distribution of all notes conditioned notes before to generate music:

$$\frac{1}{T}\sum P(v_t|v_d), d < t-1$$

where $v_t$ is the note vector on time t.

In addition to modeling this joint distribution, we also design the network to capture the patterns and structures that are invariant along translations along the note axis. Formally, given $v_i^{(t)}$, $v_i^{(t+1)}$, $w_i^{(t)}$, $w_i^{(t+1)}$

$$P(v_i^{(t+1)}|v_i^{(t)}) = P(w_i^{(t+1)}|w_i^{(t)})$$

where $v_i^{(t)}$ is $w_i^{(t)}$ shifted forward along the note axis by d positions, so $v_i^{(t+1)} = w_{i+d}^{(t+1)}$.This means that given two note shifted variant vectors, the network should produce the similarly shifted distribution of outputs, which is an extremely important property of music allowing it to be played in different keys and scales.

To learn these relative structures along both the time and note dimensions, a stack of LSTM cells is applied recurrently along each dimension. Thus the biaxial network manifests as a two-tiered architecture, where each tier independently learns the joint sequential distribution along an axis. Since music often contains varying ranges of dependencies, LSTM networks arise as a natural fit for the task due to their invariant nature across the sequential axis, as well as their improved ability to learn long range dependencies during training, thanks to their non-multiplicative errors.

### 1.0.4 Implementation

Here we need to train our model. As mentioned before, our model is a two-tiered combination of LSTMS along the time and note axis. In each step, the note-axis LSTM layers receive as input of two sources: the activations of the final time-axis LSTM layer for this note, and the final output of the network for the previous note step. The final activations of the note-axis LSTMs will be transformed into a row of two probabilities $p_0^{n,t}$, $p_1^{n,t}$ by the sigmoid function, where $p_0^{n,t}$ is the probability of note n being play at time t , and $p_1^{n,t}$ is the probability of it being articulated.

Our Our loss is then given by the negative log likelihood:

$$loss = -\frac{1}{2TN}\sum_{t=1}^{T}\sum_{n=1}^{N}(log(p_0^{n,t}x_0^{n,t} + (1-p_0^{n,t})(1-x_0^{n,t})) + log(p_1^{n,t}x_1^{n,t} + (1-p_1^{n,t})(1-x_1^{n,t}))x_0^{n,t})$$

when the note is not played, we can ignore $p_1^{n,t}$

**Step1: Import Package and Function** Firstly, we need to import functions, which helps us to clean data and train models. And we also need packages, like tensorflow, for us to train our model.

```
In [ ]: import os
        os.chdir('../lib')

        import IPython

        import model_tb as model
        import data


        import pickle
        import sys
        import tensorflow as tf
        import argparse
```

**Step2: Train Model** We need to know whetheher this model is pre-trained or not. We will check for the pre-trained model, if not, the model will be trained by our model.biaxial_model function. In this step, we generated two models: classical-music model and pop-music model.

**train on classical music**

```
In [ ]: cache_name = '../output/cache.pkl'
        model_name = None


        if not os.path.exists(cache_name):
            composers = []
            all_pieces = {}

            if len(composers)==0:
                all_pieces.update(data.getpices(path="../data/midis", mode='all'))
            elif composers == 'pop':
                all_pieces.update(data.getpices(path='../data/pop_midis', mode='all'))
            else:
                for c in composers:
                    all_pieces.update(data.getpices(path="../data/midis", composer=c))

            cache = data.initialize_cache(all_pieces, save_loc=cache_name)
        else:
            with open(cache_name, 'rb') as f:
                cache = pickle.load(f)

        # Build and load the pre-existing model if it exists
        print('Building model')
        music_model = model.biaxial_model(t_layer_sizes=[300,300],
            n_layer_sizes=[100,50],
            trainer = tf.train.AdamOptimizer())

        print('Start training')
        music_model.train(cache, batch_size=5, max_epoch=50000,
            predict_freq=100, pre_trained_model=model_name,save_freq=1)
```

**train on pop music**

```
In [ ]: cache_name = '../output/pop_cache.pkl'
        model_name = None


        if not os.path.exists(cache_name):
            composers = 'pop'
            all_pieces = {}

            if len(composers)==0:
```

```
            all_pieces.update(data.getpices(path="../data/midis", mode='all'))
        elif composers == 'pop':
            all_pieces.update(data.getpices(path='../data/pop_midis', mode='all'))
        else:
            for c in composers:
                all_pieces.update(data.getpices(path="../data/midis", composer=c))

        cache = data.initialize_cache(all_pieces, save_loc=cache_name)
    else:
        with open(cache_name, 'rb') as f:
            cache = pickle.load(f)

    # Build and load the pre-existing model if it exists
    print('Building model')
    music_model = model.biaxial_model(t_layer_sizes=[300,300],
        n_layer_sizes=[100,50],
        trainer = tf.train.AdamOptimizer())

    print('Start training')
    music_model.train(cache, batch_size=5, max_epoch=50000,
        predict_freq=100, pre_trained_model=model_name)
```

**Step3: Predict** We first get an initializer seed, sampled from the starting timestep of a measure in a random piece, as the last time step information. This input is fed through one timestep in the time network and then through the entire note dimension recurrently. The output is a complete "chord" distribution for one timestep. This output is then mapped to a new set of input vectors for each note, which is then processed through the biaxial network in a similar fashion. Once a sequence of chords with the desired length has been produced, the outputs are concatenated along the time dimension.

The pre-trained models name is biaxial_rnn_1524342232, which is trainned by classical music. We can use this model to generated music

```
In [ ]: cache_name = '../output/pop_cache.pkl'
        model_name = '../output/model/biaxial_rnn_1524342232' # this model is pre-trained on cla


        if not os.path.exists(cache_name):
            composers = []
            all_pieces = {}

            if len(composers)==0:
                all_pieces.update(data.getpices(path="../data/midis", mode='all'))
            elif composers == 'pop':
                all_pieces.update(data.getpices(path='../data/pop_midis', mode='all'))
            else:
                for c in composers:
                    all_pieces.update(data.getpices(path="../data/midis", composer=c))
```

```
    cache = data.initialize_cache(all_pieces, save_loc=cache_name)
else:
    with open(cache_name, 'rb') as f:
        cache = pickle.load(f)

# Building model
print('Building model')
music_model = model.biaxial_model(t_layer_sizes=[300,300], n_layer_sizes=[100,50],
                                  trainer=tf.train.AdamOptimizer())

print('Start predicting')
music_model.predict(cache,model_name,step=320,conservativity=1,n=20,saveto='../output/pr
```

#### 1.0.5 Result

We trained two models, using classical music and pop music separately. The following selected music is generated by our model.

```
In [ ]: IPython.display.Audio('../doc/classical.mp3')
```

```
In [ ]: IPython.display.Audio('../doc/classical_guitar.mp3')
```

```
In [ ]: IPython.display.Audio('../doc/pop.mp3')
```

It is not hard to see that these music have different styles, which is determined by the trained music.To some degree, the result of pop-music model is more melodious than classical-music model, according to the complexity of trained music. However, surprisingly, our classical-music model can play chords. Although the melody is not so mellifluence. But still the generated musice from two model are quite interesting.