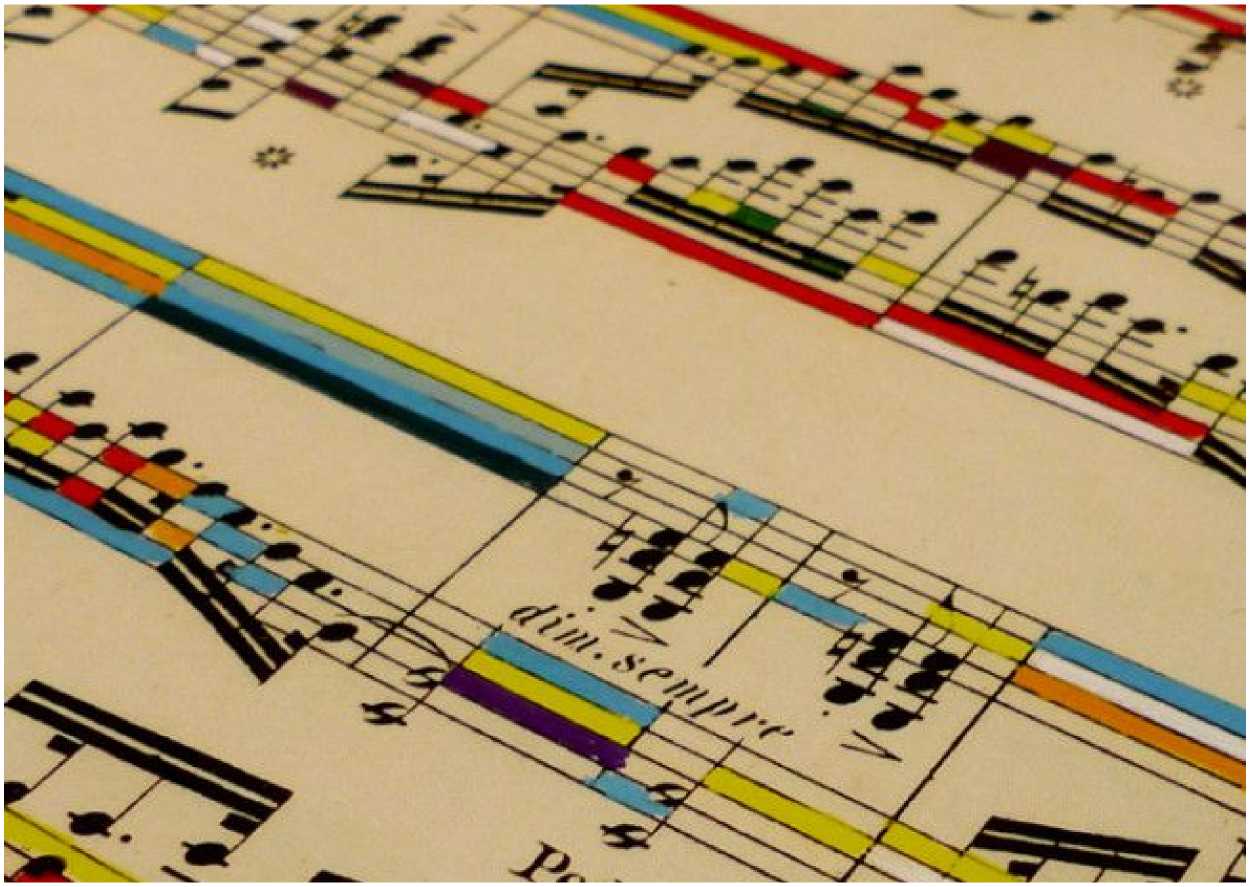
INDIVIDUAL STUDY REPORT

GENERATING MUSIC USING TENSORFLOW



ANUVRAT TIKU

010822084

CMPE-297

MACHINE LEARNING

PROFESSOR : REX TSOU

TABLE OF CONTENTS

ABSTRACT

TensorFlow is an open source Machine Learning library for numerical computations. Dat flow is done using multidimensional arrays called Tensors and these tensors flow between graph nodes via edges. TensorFlow was developed by Google and open sourced. It’s build using C++ and API integration and programming is done using Python. My attempt will be to generate music, which would basically be an extension of sound with notes using TensorFlow.

For this purpose, I will be using a Neural Net called the RBM or the Restricted Boltzmann Machine. RBMs are shallow, two-layer neural nets that constitute the building blocks of deep-belief networks. The first layer of the RBM is called the visible, or input, layer, and the second is the hidden layer and there is no communication between two nodes in the same layer. For the training part, I will use the Gibbs Sampling which is an effective music generation technique. Gibbs sampling helps us create audio samples during training by randomly obtaining them from a generated probability distribution based on input data.

Libraries used : TensorFlow, Pandas, Numpy

BACKGROUND

HISTORICAL SIGNIFICANCE AND CHALLENGE

History behind computer generated music :

Google’s DeepMind released a paper quite a while ago called Wavenet. Wavenet is the state of the art in both music generation and text-to-speech. Traditionally audio generation models are concatenated that means in order to generate speech from some text sample, it requires a huge database of speech fragments by picking a few and combining them to create a full sentence, same for music. This works but it's hard to incorporate things like emotion and natural sound when the output is created by putting a bunch of static fragments together. Ideally we can have audio generation that is parametric where all the information required to generate audio is stored in the parameters of the model that we give it. That's what Wavenet is capable of. Instead of generating an audio signal bypassing the output through signal processing algorithms. It directly models the raw waveform of the audio signals. Researchers don't really do that but wavenet did. The model they use with a convolutional neural network where each layer had dilation factors that lets its interconnectedness grow exponentially the deeper the data flow through the model and each generated sample at each step was fed back into the network to generate the next step. Let's understand the computation graph for the model.

The input data a single node starts as a raw audio wave. We first format the wave so that it's better suited for processing. Then we encode it to produce a tensor with a number of samples and a number of channels. We feed that into the first layer of the convolutional network which reduces the number of channels for easier processing and through a stack of layers with dilated convolution we combine the outputs of all the layers and increase the dimensionality to the original number of channels and feed it all to a loss function which measures how well our training is going and finally the output is fed back into the network to generate the wave at the next time step. This process keeps repeating to generate more and more audio. This neural net was huge it took 90 minutes on their GPU cluster to generate a single second of audio.

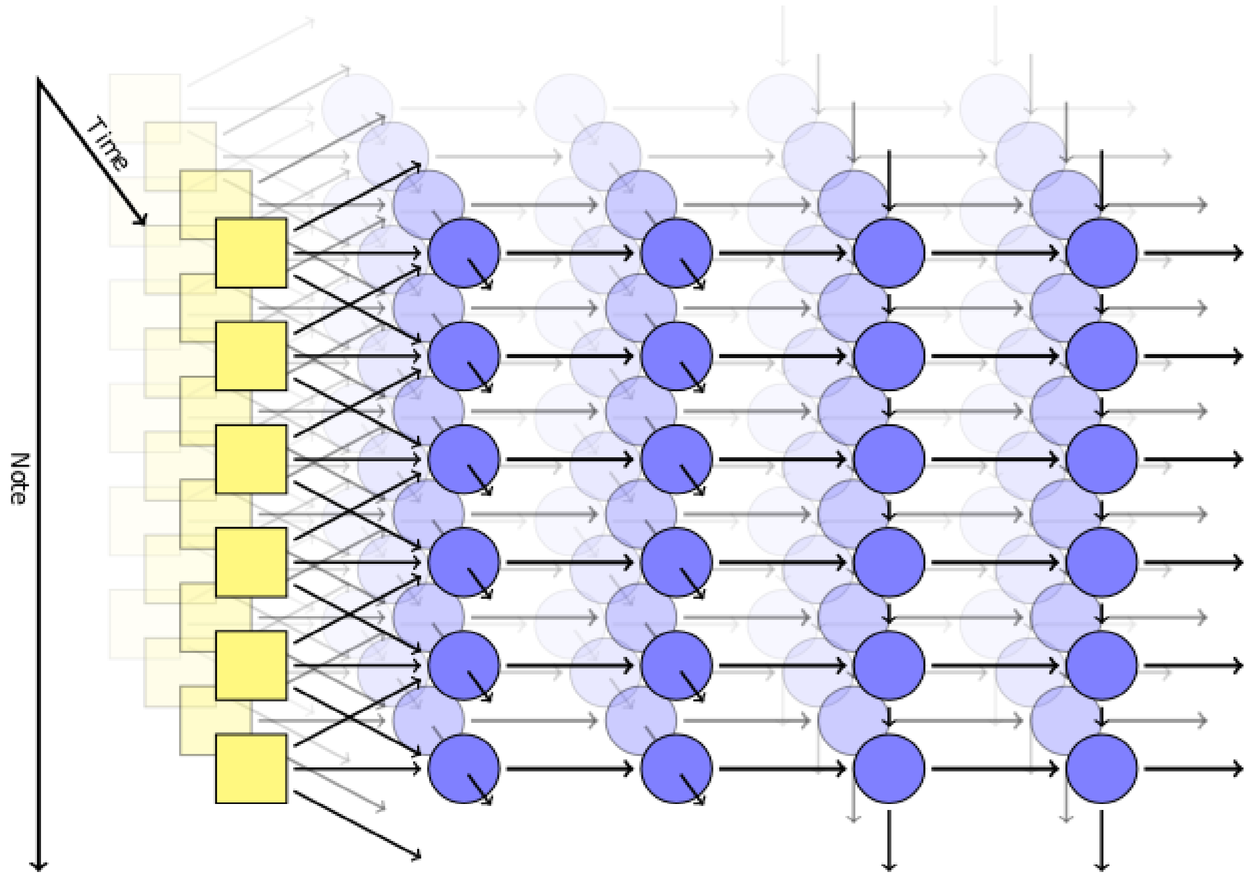


Fig 1.1 : Restricted Boltzmann Machine

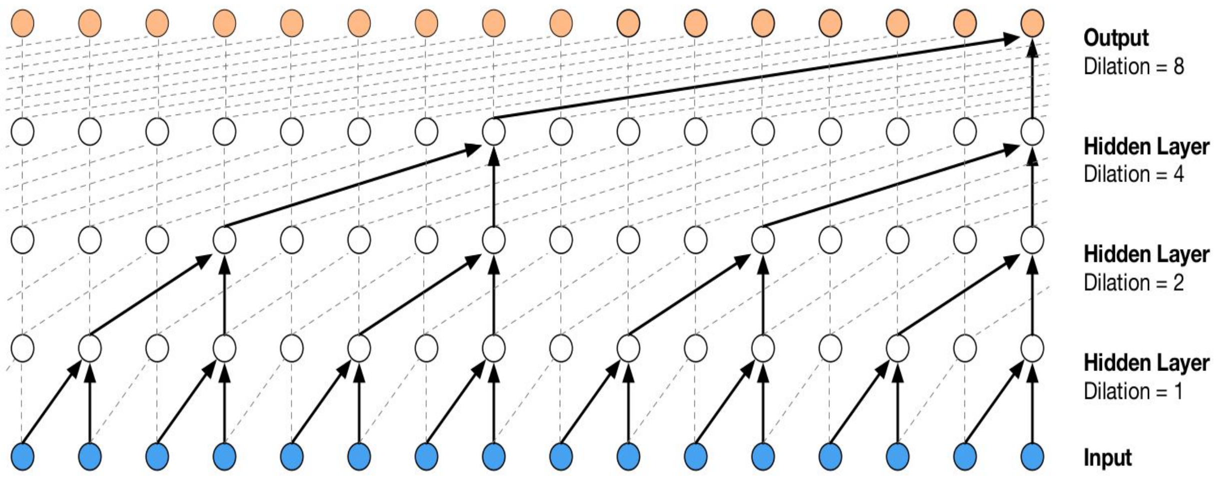


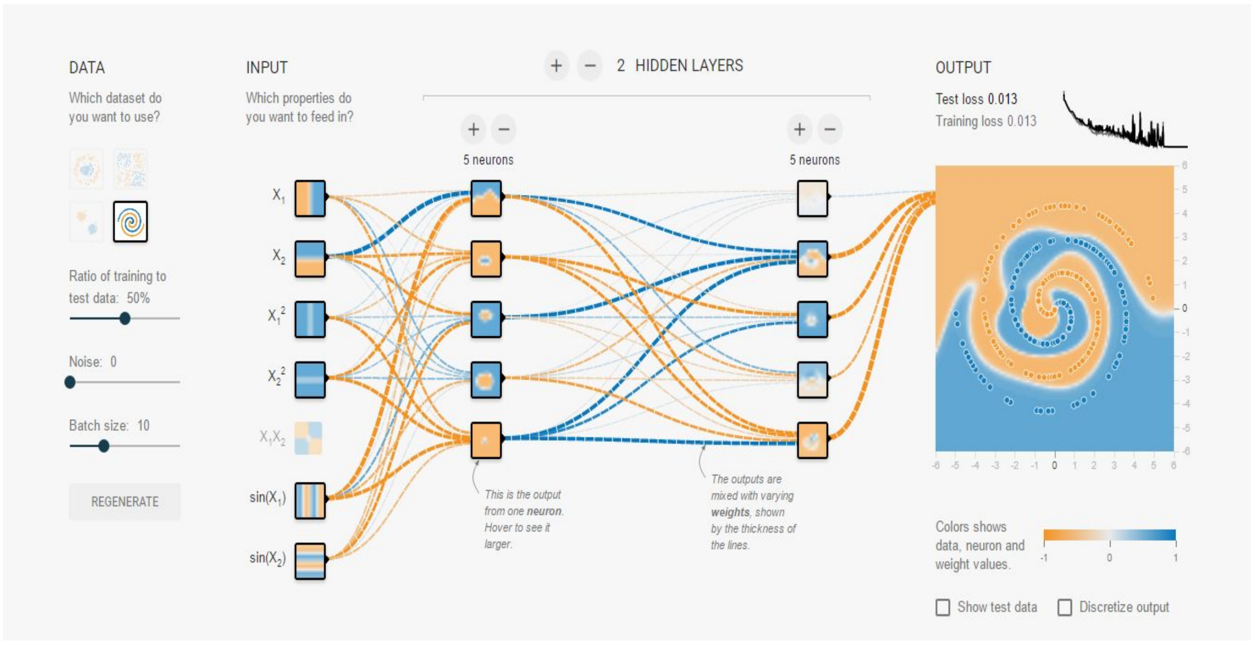
Fig 1.2 : Flow of data through a neural net with hidden layers like a RBM

SUMMARY

Instead of following the Wavenet approach, we will be suing a much simpler approach towards music generation using TensorFlow. The Wavenet method requires significant CPU and GPU computation power and as it was pointed out above, it took took the google servers more than 90 minutes to generate a second of music. Our approach centers around the use of Google's library called Tensorflow.

TensorFlow :

TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.



Ratio of training to test data: 50%

Noise: 0

Batch size: 10

Test loss 0.013

Training loss 0.013

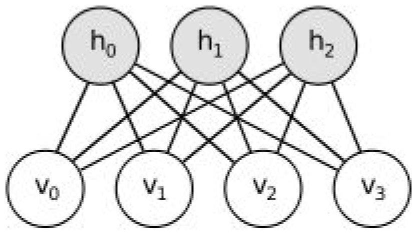
This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Fig 1.3 : TensorFlow - A neural network playground.

Boltzmann Machine (BM) :

Boltzmann Machines (BM) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. An typical Boltzmann Machine is displayed below :



TECHNICAL APPROACH

PREPARATION FOR EXECUTION

Deviating from Wavenet approach, instead we're going to build a more simple music generation script in tensorflow. We want to first important numpy which is a scientific computing library in Python and pandas are data analytics library then of course tensorflow our machine learning library t2dm will help us show a progress bar during training and finally MIDI manipulation which is our helper library let's start off with the first step are hyper parameters or tuning knobs for our model.

We're going to use the type of neural network called a restricted Boltzmann machine as our generative model. As described earlier, a RBM is a two-layer neural net, in which the first layer is visible and the next layer is hidden. Nodes within layers are connected to note in the next layer but no two nodes of the same layer are linked : that's the restriction part. Each node decides whether to transmit any data it receives to the next layer or not by making a random decision. We want to first create and range of notes that our model generates so we will write variables for our lowest note and our highest note as well as a note range which is the difference between the two. Then we'll write variables for the number of times(timestamp), the size of the visible layers, number of hidden layers, number of training epochs, bats eyes and finally our learning rate which is a tensorflow constant since it always stays the same.

Next we will start out by writing variables. We will start by creating a placeholder which will serve as our gateway for inputting data into our model, then a variable that will store the weight matrix or the connections between our two layers. We also need two variables for biases, one for the hidden layer and one for the visible area. We then want to create a sample from our input in X-axis using Gibbs sample helper method.

A little about Gibbs sampling

Gibbs sampling is an algorithm that creates a sample from a multivariate probability distribution. It construct a statistical model for each state depends on the previous state and then uses randomness to obtain a sample across the distribution it creates. We will get a sample of the hidden nodes as well starting from our original input and simple of the hidden node starting from are generated sample. Then we will update our weight values based on the differences between the samples that we drew and the original values. We will run these three update steps for our weights, hidden biases and visible biases when we later our session and we can store all of these in our update variable.

NEURAL NETS

Neural Networks (also referred to as connectionist systems) are a computational approach which is based on a large collection of neural units loosely modeling the way the brain solves problems with large clusters of biological neurons connected by axons. Each neural unit is connected with many others, and links can be enforcing or inhibitory in their effect on the activation state of connected neural units. Each individual neural unit may have a summation function which combines the values of all its inputs together. There may be a threshold function or limiting function on each connection and on the unit itself such that it must surpass it before it can propagate to other neurons. These systems are self-learning and trained rather than explicitly programmed and excel in areas where the solution or feature detection is difficult to express in a traditional computer program.

PANDAS

Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas is a NUMFocus sponsored project. This will help ensure the success of development of pandas as a world-class open-source project.

What can pandas do : Python has long been great for data munging and preparation, but less so for data analysis and modeling. pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R. Combined with the excellent IPython toolkit and other libraries, the environment for doing data analysis in Python excels in performance, productivity, and the ability to collaborate.

NUMPY

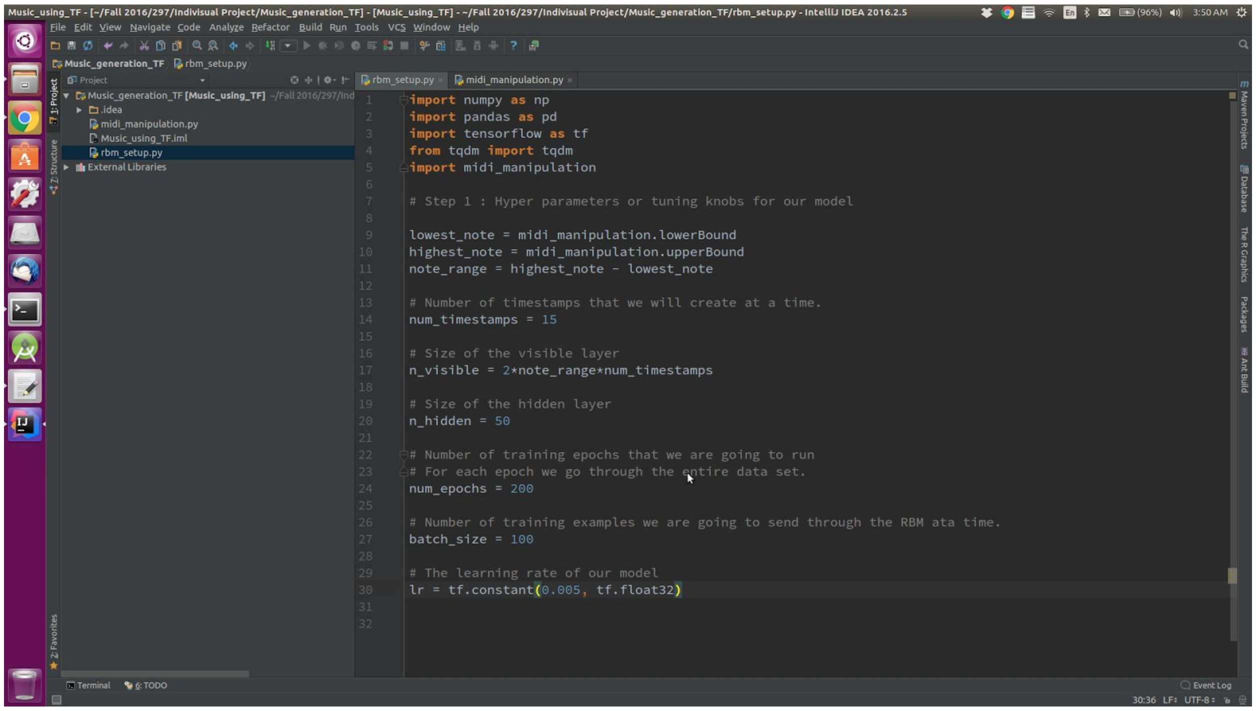
NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

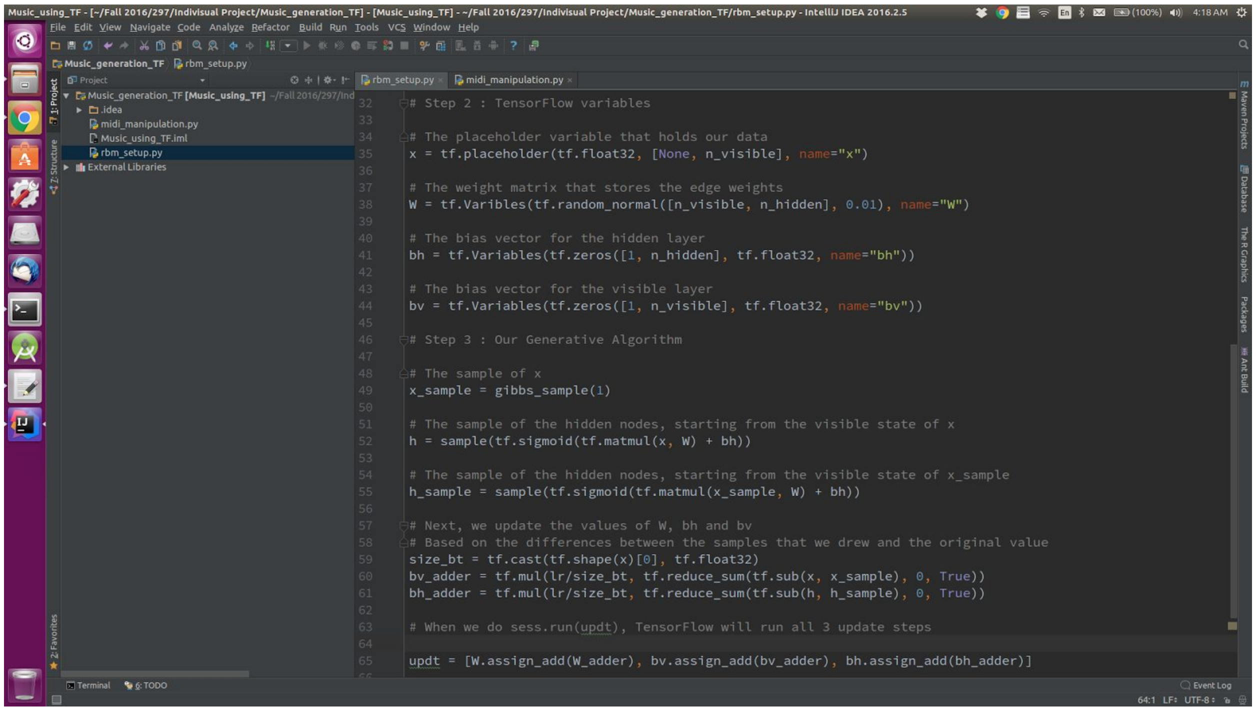
We are all done with the setup. Now it’s time to start the session and run out program.

Initializing RBM code to generate music in 4 steps



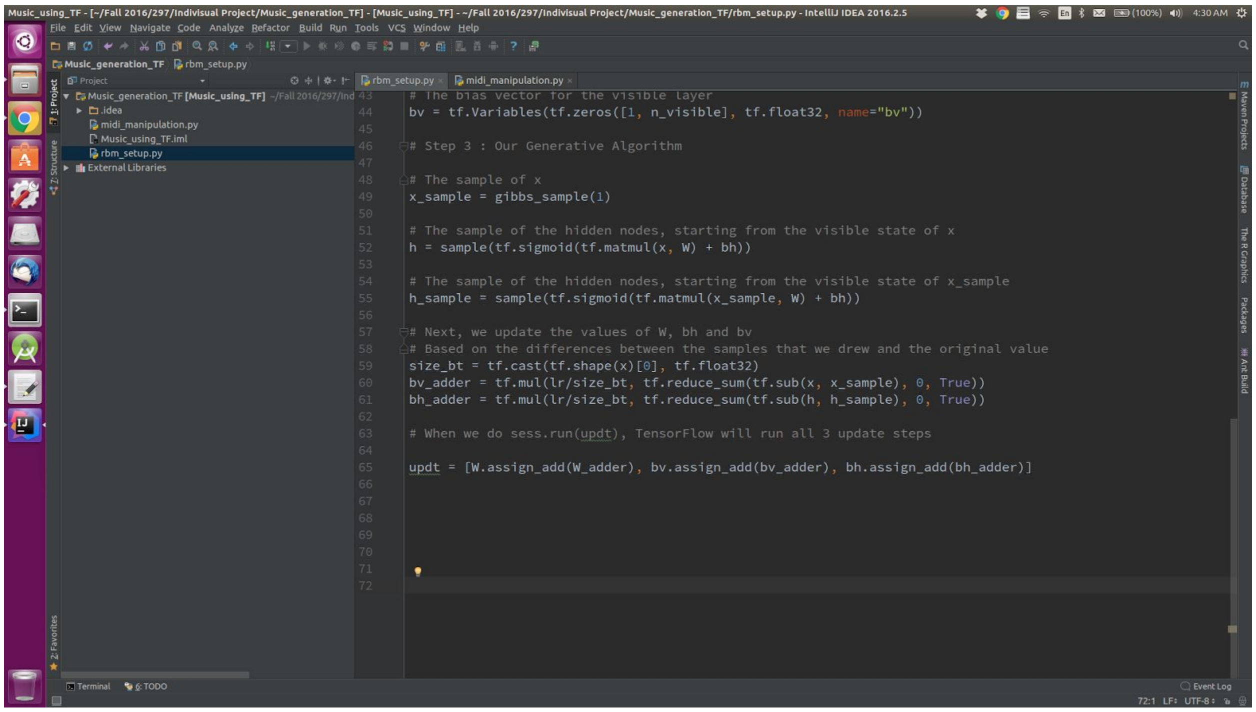
```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tqdm import tqdm
import midi_manipulation

# Step 1 : Hyper parameters or tuning knobs for our model

lowest_note = midi_manipulation.lowerBound
highest_note = midi_manipulation.upperBound
note_range = highest_note - lowest_note

# Number of timestamps that we will create at a time.
num_timestamps = 15

# Size of the visible layer
n_visible = 2*note_range*num_timestamps

# Size of the hidden layer
n_hidden = 50

# Number of training epochs that we are going to run
# For each epoch we go through the entire data set.
num_epochs = 200

# Number of training examples we are going to send through the RBM ata time.
batch_size = 100

# The learning rate of our model
lr = tf.constant(0.005, tf.float32)
```

Fig 1.4 : Setting up or RBM model hyper parameters or tuning knobs



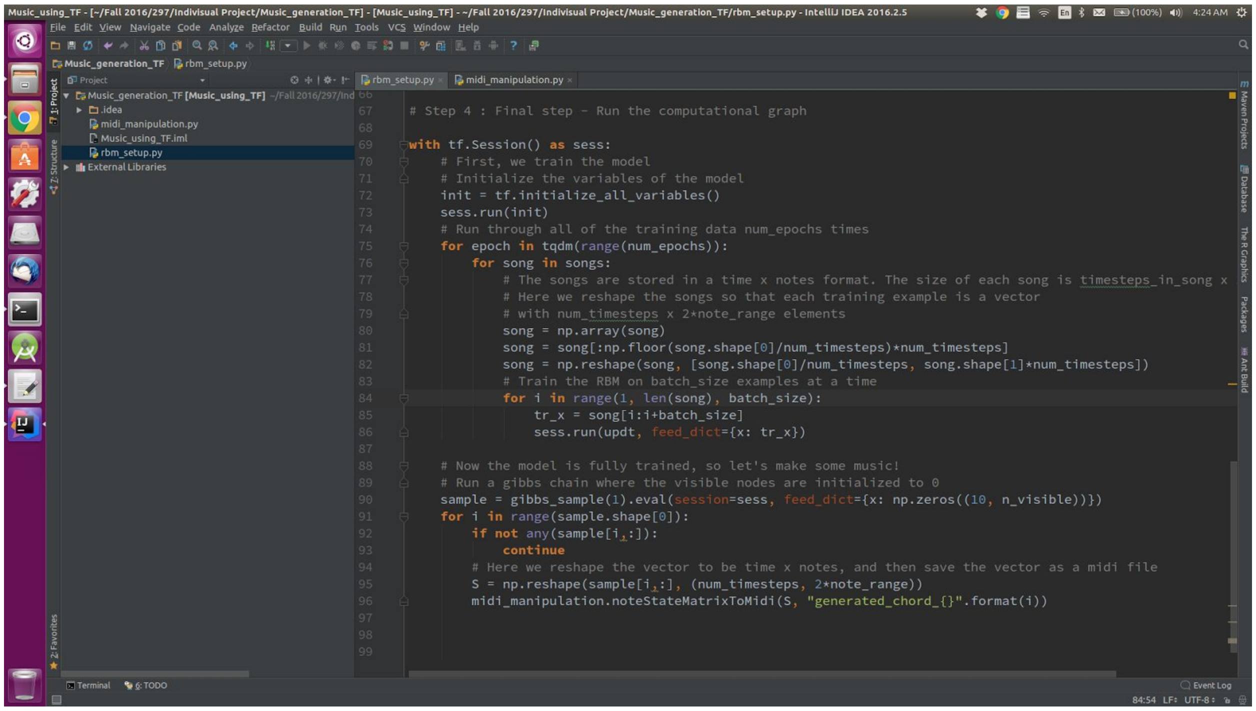
```python
# Step 2 : TensorFlow variables

# The placeholder variable that holds our data
x = tf.placeholder(tf.float32, [None, n_visible], name="x")

# The weight matrix that stores the edge weights
W = tf.Varibles(tf.random_normal([n_visible, n_hidden], 0.01), name="W")

# The bias vector for the hidden layer
bh = tf.Variables(tf.zeros([1, n_hidden], tf.float32, name="bh"))

# The bias vector for the visible layer
bv = tf.Variables(tf.zeros([1, n_visible], tf.float32, name="bv"))

# Step 3 : Our Generative Algorithm

# The sample of x
x_sample = gibbs_sample(1)

# The sample of the hidden nodes, starting from the visible state of x
h = sample(tf.sigmoid(tf.matmul(x, W) + bh))

# The sample of the hidden nodes, starting from the visible state of x_sample
h_sample = sample(tf.sigmoid(tf.matmul(x_sample, W) + bh))

# Next, we update the values of W, bh and bv
# Based on the differences between the samples that we drew and the original value
size_bt = tf.cast(tf.shape(x)[0], tf.float32)
bv_adder = tf.mul(lr/size_bt, tf.reduce_sum(tf.sub(x, x_sample), 0, True))
bh_adder = tf.mul(lr/size_bt, tf.reduce_sum(tf.sub(h, h_sample), 0, True))

# When we do sess.run(updt), TensorFlow will run all 3 update steps

updt = [W.assign_add(W_adder), bv.assign_add(bv_adder), bh.assign_add(bh_adder)]
```

Fig 1.5 : Declaring TensorFlow variables


Music_using_TF - [~/Fall 2016/297/Indivisual Project/Music_generation_TF] - [Music_using_TF] - ~/Fall 2016/297/Indivisual Project/Music_generation_TF/rbm_setup.py - IntelliJ IDEA 2016.2.5
Music_generation_TF
rbm_setup.py
midi_manipulation.py
Music_using_TF.iml
External Libraries
Terminal
6: TODO
Event Log

```
# The bias vector for the visible layer
bv = tf.Variables(tf.zeros([1, n_visible], tf.float32, name="bv"))

# Step 3 : Our Generative Algorithm

# The sample of x
x_sample = gibbs_sample(1)

# The sample of the hidden nodes, starting from the visible state of x
h = sample(tf.sigmoid(tf.matmul(x, W) + bh))

# The sample of the hidden nodes, starting from the visible state of x_sample
h_sample = sample(tf.sigmoid(tf.matmul(x_sample, W) + bh))

# Next, we update the values of W, bh and bv
# Based on the differences between the samples that we drew and the original value
size_bt = tf.cast(tf.shape(x)[0], tf.float32)
bv_adder = tf.mul(lr/size_bt, tf.reduce_sum(tf.sub(x, x_sample), 0, True))
bh_adder = tf.mul(lr/size_bt, tf.reduce_sum(tf.sub(h, h_sample), 0, True))

# When we do sess.run(updt), TensorFlow will run all 3 update steps

updt = [W.assign_add(W_adder), bv.assign_add(bv_adder), bh.assign_add(bh_adder)]
```

Fig 1.6 : Our generative algorithm


Music_using_TF - [~/Fall 2016/297/Indivisual Project/Music_generation_TF] - [Music_using_TF] - ~/Fall 2016/297/Indivisual Project/Music_generation_TF/rbm_setup.py - IntelliJ IDEA 2016.2.5
Music_generation_TF
rbm_setup.py
midi_manipulation.py
Music_using_TF.iml
External Libraries
Terminal
6: TODO
Event Log

```
# Step 4 : Final step - Run the computational graph

with tf.Session() as sess:
    # First, we train the model
    # Initialize the variables of the model
    init = tf.initialize_all_variables()
    sess.run(init)
    # Run through all of the training data num_epochs times
    for epoch in tqdm(range(num_epochs)):
        for song in songs:
            # The songs are stored in a time x notes format. The size of each song is timesteps_in_song x
            # Here we reshape the songs so that each training example is a vector
            # with num_timesteps x 2*note_range elements
            song = np.array(song)
            song = song[:np.floor(song.shape[0]/num_timesteps)*num_timesteps]
            song = np.reshape(song, [song.shape[0]/num_timesteps, song.shape[1]*num_timesteps])
            # Train the RBM on batch_size examples at a time
            for i in range(1, len(song), batch_size):
                tr_x = song[i:i+batch_size]
                sess.run(updt, feed_dict={x: tr_x})

    # Now the model is fully trained, so let's make some music!
    # Run a gibbs chain where the visible nodes are initialized to 0
    sample = gibbs_sample(1).eval(session=sess, feed_dict={x: np.zeros((10, n_visible))})
    for i in range(sample.shape[0]):
        if not any(sample[i,:]):
            continue
        # Here we reshape the vector to be time x notes, and then save the vector as a midi file
        S = np.reshape(sample[i,:], (num_timesteps, 2*note_range))
        midi_manipulation.noteStateMatrixToMidi(S, "generated_chord_{}".format(i))
```

Fig 1.7 : Running the computational graph

EXECUTION

Alright it's time to start session so we can run our graph.

1. We will initialize our variables first. We're going to reshape each song so they are vectorize in a good training format.
2. Then we're going to train our RBM on the examples, one at a time.
3. Once the model is fully trained we will make some music. We run our gibbs chain with a visible nodes are initialized to 0 to generate a sample and reshape the vector to be the proper format for playback.
4. Finally will print out the generated chords.

ISSUES AND CONTROVERSIES

OVERFITTING :

To prevent our model from being overfit (which would mean learning specific parts of specific pieces instead of overall patterns and features), we can use something called dropout. Applying dropout essentially means randomly removing half of the hidden nodes from each layer during each training step. This prevents the nodes from gravitating toward fragile dependencies on each other and instead promotes specialization. (We can implement this by multiplying a mask with the outputs of each layer. Nodes are "removed" by zeroing their output in the given time step.)

USE OF ANALYTICS LIBRARY - PANDAS :

Pandas does not implement significant modeling functionality outside of linear and panel regression; for this, look to statsmodels and scikit-learn. More work is still needed to make Python a first class statistical modeling environment, but we are well on our way toward that goal.

PERFORMANCE ISSUES - TRAINING :

We can make training faster by taking advantage of the fact that we already know exactly which output we will choose at each time step. Basically, we can first batch all of the notes together and train the time-axis layers, and then we can reorder the output to batch all of the times together and train all the note-axis layers. This allows us to more effectively utilize the GPU, which is good at multiplying huge matrices.

DISCUSSION

MY TAKE ON THE SUBJECT

1. IMPLEMENTATION - NEURAL NETS :

During composition, we unfortunately cannot batch everything as effectively. At each time step, we have to first run the time-axis layers by one tick, and then run an entire recurrent sequence of the note-axis layers to determine what input to give to the time-axis layers at the next tick. This makes composition slower. In addition, we have to add a correction factor to account for the dropout during training. Practically, this means multiplying the output of each node by 0.5. This prevents the network from becoming over excited due to the higher number of active nodes.

2. TRAINING :

We can make training faster by taking advantage of the fact that we already know exactly which output we will choose at each time step. Basically, we can first batch all of the notes together and train the time-axis layers, and then we can reorder the output to batch all of the times together and train all the note-axis layers. This allows us to more effectively utilize the GPU, which is good at multiplying huge matrices.

3. CONVOLUTION IN NEURAL NETWORKS :

There is one kind of neural network that is widely in use today that has this invariant property along multiple directions: convolutional neural networks for image recognition. These work by basically learning a convolution kernel and then applying that same convolution kernel across every pixel of the input image.

Hypothetically, what would happen if we replaced the convolution kernel with something else? Say, a recurrent neural network? Then each pixel would have its own neural network, which would take input from an area around the pixel. Each neural network would in turn have its own memory cells and recurrent connections across time. Now replace pixels with notes, and we have an idea for what we can do. If we make a stack of identical recurrent neural networks, one for each output note, and give each one a local neighborhood (for example, one octave above and below) around the note as its input, then we have a system that is invariant in both time and notes: the network can work with relative inputs in both directions.

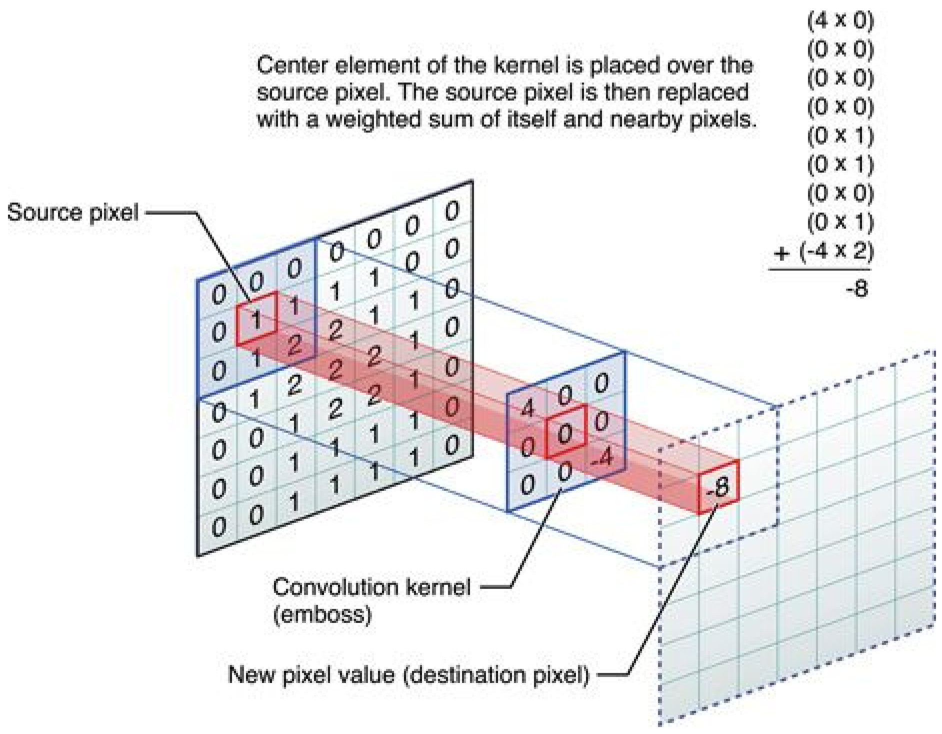

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Fig 1.8 : An ideal Convoluted Neural Network for image recognition

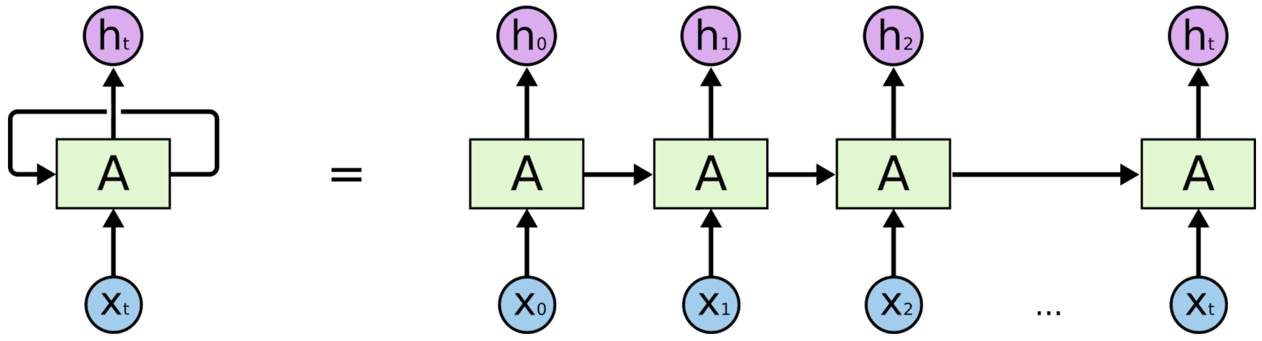


Fig 1.9 : A RNN based on the above CNN used to generate music in machines.

CONCLUSION

Note the state of the art and music generation uses a CNN to generate raw waveforms parametrically instead of concatenate flee that algorithm requires lots of computer right now so we can use an RBM to generate audio samples based on training data easily and gibbs sampling helps us create audio samples during training by randomly obtaining them from a generated probability distribution based on the input data.

Music can be easily generated using RBM.

Gibbs sampling can be used as an effective music generation technique.

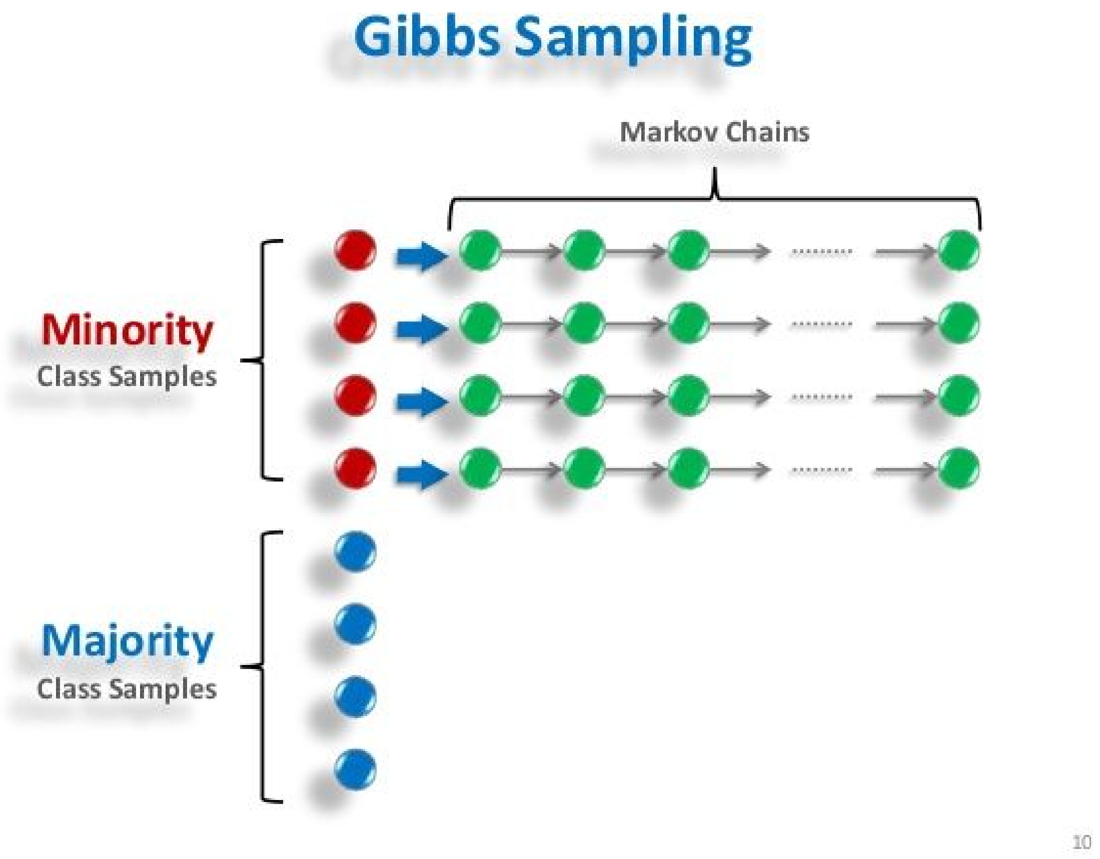


Fig 1.10 : Gibbs sampling for sampling Markov Chains

The implementation of the code :
https://github.com/anuvrat-tiku/Music_Generation_TensorFlow

Sample Audio Clips Generated :
https://www.dropbox.com/sh/66dilg0dhaike3q/AAB85tbeVBSyAe4Eiktcz3LTa?dl=0

REFERENCES

[1] Ilker Yildirim, Bayesian Inference: Gibbs Sampling, MIT Press, August 2012

[2] RBM, https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine

[3] Deep Learning, http://deeplearning.net/

[4] TensorFlow, https://www.tensorflow.org/versions/r0.11/tutorials/index.html

[5] Back Propogation,
http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

[6] Asja Fischer, Christian Igel, Markov Chains and Unsupervised learning,
http://image.diku.dk/igel/paper/AItRBM-proof.pdf

[7] Pandas, Data Analytics, http://pandas.pydata.org/pandas-docs/stable/tutorials.html