```
7     54 data.py
8     23 LICENSE.txt
2     48 main.py
9    100 midi_to_statematrix.py
     26 midi_to_statematrix.pyc
3    403 model.py
10     61 multi_training.py
     21 multi_training.pyc
11     18 out_to_in_op.py
      0 __pycache__
1     57 readme.md
4      7 setup1.sh
 5    11 setup2.sh
  6   34 setup_optional.sh
12     72 visualize.py
    935 total
```

Biaxial Recurrent Neural Network for Music Composition

This code implements a recurrent neural network trained to generate classical music. The model, which uses LSTM layers and draws inspiration from convolutional neural networks, learns to predict which notes will be played at each time step of a musical piece.

You can read about its design and hear examples on [this blog post](http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/).

Requirements

This code is written in Python, and depends on having Theano and theano-lstm (which can be installed with pip) installed. The bare minimum you should need to do to get everything running, assuming you have Python, is
```
sudo pip install --upgrade theano
sudo pip install numpy scipy theano-lstm python-midi
```

In addition, the included setup scripts should set up the environment exactly as it was when I trained the network on an Amazon EC2 g2.2xlarge instance with an external EBS volume. Installing it with other setups will likely be slightly different.

Using it

First, you will need to obtain a large selection of midi music, preferably in 4/4 time, with notes correctly aligned to beats. These can be placed in a directory "music".

To use the model, you need to first create an instance of the Model class:
```python
import model
m = model.Model([300,300],[100,50], dropout=0.5)
```

where the numbers are the sizes of the hidden layers in the two parts of the network architecture. This will take a while, as this is where Theano will compile its optimized functions.

Next, you need to load in the data:
```python
import multi_training
pcs = multi_training.loadPieces("music")
```

Then, after creating an "output" directory for trained samples, you can start training:
```python
multi_training.trainPiece(m, pcs, 10000)
```

This will train using 10000 batches of 10 eight-measure segments at a time, and output a sampled output and the learned parameters every 500 iterations.

Finally, you can generate a full composition after training is complete. The function `gen_adaptive` in main.py will generate a piece and also prevent long empty gaps by increasing note probabilities if the network stops playing for too long.
```python
gen_adaptive(m,pcs,10,name="composition")
```

There are also mechanisms to observe the hidden activations and memory cells of the network, but these are still a work in progress at the moment.

Right now, there is no separate validation step, because my initial goal was to produce interesting music, not to assess the accuracy of this method. It does, however, print out the cost on the training set after every 100 iterations during training.

If you want to save your model weights, you can do
```python
pickle.dump( m.learned_config, open( "path_to_weight_file.p", "wb" ) )
```
and if you want to load them, you can do
```python
m.learned_config = pickle.load(open( "path_to_weight_file.p", "rb" ) )
```

```
import cPickle as pickle
import gzip
import numpy
from midi_to_statematrix import *

import multi_training
import model

def gen_adaptive(m,pcs,times,keep_thoughts=False,name="final"):
```

```
        xIpt, xOpt = map(lambda x: numpy.array(x, dtype='int8'),
 multi_training.getPieceSegment(pcs))
        all_outputs = [xOpt[0]]
        if keep_thoughts:
                all_thoughts = []
        m.start_slow_walk(xIpt[0])
        cons = 1
        for time in range(multi_training.batch_len*times):
                resdata = m.slow_walk_fun( cons )
                nnotes = numpy.sum(resdata[-1][:,0])
                if nnotes < 2:
                        if cons > 1:
                                cons = 1
                        cons -= 0.02
                else:
                        cons += (1 - cons)*0.3
                all_outputs.append(resdata[-1])
                if keep_thoughts:
                        all_thoughts.append(resdata)
        noteStateMatrixToMidi(numpy.array(all_outputs),'output/'
+name)
        if keep_thoughts:
                pickle.dump(all_thoughts, open('output/'+name+'.
p','wb'))

def fetch_train_thoughts(m,pcs,batches,name="trainthoughts"):
        all_thoughts = []
        for i in range(batches):
                ipt, opt = multi_training.getPieceBatch(pcs)
                thoughts = m.update_thought_fun(ipt,opt)
                all_thoughts.append((ipt,opt,thoughts))
        pickle.dump(all_thoughts, open('output/'+name+'.p','wb')
)

if __name__ == '__main__':

        pcs = multi_training.loadPieces("music")

        m = model.Model([300,300],[100,50], dropout=0.5)

        multi_training.trainPiece(m, pcs, 10000)

        pickle.dump( m.learned_config, open( "output/final_learn
ed_config.p", "wb" ) )
import theano, theano.tensor as T
import numpy as np
import theano_lstm

from out_to_in_op import OutputFormToInputFormOp

from theano_lstm import Embedding, LSTM, RNN, StackedCells, Laye
r, create_optimization_updates, masked_loss, MultiDropout

def has_hidden(layer):
    """
    Whether a layer has a trainable
```

```
    initial hidden state.
    """
    return hasattr(layer, 'initial_hidden_state')

def matrixify(vector, n):
    # Cast n to int32 if necessary to prevent error on 32 bit sy
stems
    return T.repeat(T.shape_padleft(vector),
                    n if (theano.configdefaults.local_bitwidth()
 == 64) else T.cast(n,'int32'),
                    axis=0)

def initial_state(layer, dimensions = None):
    """
    Initalizes the recurrence relation with an initial hidden st
ate
    if needed, else replaces with a "None" to tell Theano that
    the network **will** return something, but it does not need
    to send it to the next step of the recurrence
    """
    if dimensions is None:
        return layer.initial_hidden_state if has_hidden(layer) e
lse None
    else:
        return matrixify(layer.initial_hidden_state, dimensions)
 if has_hidden(layer) else None

def initial_state_with_taps(layer, dimensions = None):
    """Optionally wrap tensor variable into a dict with taps=[-1
]"""
    state = initial_state(layer, dimensions)
    if state is not None:
        return dict(initial=state, taps=[-1])
    else:
        return None

class PassthroughLayer(Layer):
    """
    Empty "layer" used to get the final output of the LSTM
    """

    def __init__(self):
        self.is_recursive = False

    def create_variables(self):
        pass

    def activate(self, x):
        return x

    @property
    def params(self):
        return []

    @params.setter
    def params(self, param_list):
```

```
        pass


def get_last_layer(result):
    if isinstance(result, list):
        return result[-1]
    else:
        return result

def ensure_list(result):
    if isinstance(result, list):
        return result
    else:
        return [result]


class Model(object):

    def __init__(self, t_layer_sizes, p_layer_sizes, dropout=0):

        self.t_layer_sizes = t_layer_sizes
        self.p_layer_sizes = p_layer_sizes

        # From our architecture definition, size of the notewise
 input
        self.t_input_size = 80

        # time network maps from notewise input size to various
hidden sizes
        self.time_model = StackedCells( self.t_input_size, cellt
ype=LSTM, layers = t_layer_sizes)
        self.time_model.layers.append(PassthroughLayer())

        # pitch network takes last layer of time model and state
 of last note, moving upward
        # and eventually ends with a two-element sigmoid layer
        p_input_size = t_layer_sizes[-1] + 2
        self.pitch_model = StackedCells( p_input_size, celltype=
LSTM, layers = p_layer_sizes)
        self.pitch_model.layers.append(Layer(p_layer_sizes[-1],
2, activation = T.nnet.sigmoid))

        self.dropout = dropout

        self.conservativity = T.fscalar()
        self.srng = T.shared_randomstreams.RandomStreams(np.rand
om.randint(0, 1024))

        self.setup_train()
        self.setup_predict()
        self.setup_slow_walk()

    @property
    def params(self):
        return self.time_model.params + self.pitch_model.params
```

```
    @params.setter
    def params(self, param_list):
        ntimeparams = len(self.time_model.params)
        self.time_model.params = param_list[:ntimeparams]
        self.pitch_model.params = param_list[ntimeparams:]

    @property
    def learned_config(self):
        return [self.time_model.params, self.pitch_model.params,
 [l.initial_hidden_state for mod in (self.time_model, self.pitch
_model) for l in mod.layers if has_hidden(l)]]

    @learned_config.setter
    def learned_config(self, learned_list):
        self.time_model.params = learned_list[0]
        self.pitch_model.params = learned_list[1]
        for l, val in zip((l for mod in (self.time_model, self.p
itch_model) for l in mod.layers if has_hidden(l)), learned_list[
2]):
            l.initial_hidden_state.set_value(val.get_value())

    def setup_train(self):

        # dimensions: (batch, time, notes, input_data) with inpu
t_data as in architecture
        self.input_mat = T.btensor4()
        # dimensions: (batch, time, notes, onOrArtic) with 0:on,
 1:artic
        self.output_mat = T.btensor4()

        self.epsilon = np.spacing(np.float32(1.0))

        def step_time(in_data, *other):
            other = list(other)
            split = -len(self.t_layer_sizes) if self.dropout els
e len(other)
            hiddens = other[:split]
            masks = [None] + other[split:] if self.dropout else
[]
            new_states = self.time_model.forward(in_data, prev_h
iddens=hiddens, dropout=masks)
            return new_states

        def step_note(in_data, *other):
            other = list(other)
            split = -len(self.p_layer_sizes) if self.dropout els
e len(other)
            hiddens = other[:split]
            masks = [None] + other[split:] if self.dropout else
[]
            new_states = self.pitch_model.forward(in_data, prev_
hiddens=hiddens, dropout=masks)
            return new_states

        # We generate an output for each input, so it doesn't ma
ke sense to use the last output as an input.
```

```
        # Note that we assume the sentinel start value is alread
y present
        # TEMP CHANGE: NO SENTINEL
        input_slice = self.input_mat[:,0:-1]
        n_batch, n_time, n_note, n_ipn = input_slice.shape

        # time_inputs is a matrix (time, batch/note, input_per_n
ote)
        time_inputs = input_slice.transpose((1,0,2,3)).reshape((
n_time,n_batch*n_note,n_ipn))
        num_time_parallel = time_inputs.shape[1]

        # apply dropout
        if self.dropout > 0:
            time_masks = theano_lstm.MultiDropout( [(num_time_pa
rallel, shape) for shape in self.t_layer_sizes], self.dropout)
        else:
            time_masks = []

        time_outputs_info = [initial_state_with_taps(layer, num_
time_parallel) for layer in self.time_model.layers]
        time_result, _ = theano.scan(fn=step_time, sequences=[ti
me_inputs], non_sequences=time_masks, outputs_info=time_outputs_
info)

        self.time_thoughts = time_result

        # Now time_result is a list of matrix [layer](time, batc
h/note, hidden_states) for each layer but we only care about
        # the hidden state of the last layer.
        # Transpose to be (note, batch/time, hidden_states)
        last_layer = get_last_layer(time_result)
        n_hidden = last_layer.shape[2]
        time_final = get_last_layer(time_result).reshape((n_time
,n_batch,n_note,n_hidden)).transpose((2,1,0,3)).reshape((n_note,
n_batch*n_time,n_hidden))

        # note_choices_inputs represents the last chosen note. S
tarts with [0,0], doesn't include last note.
        # In (note, batch/time, 2) format
        # Shape of start is thus (1, N, 2), concatenated with al
l but last element of output_mat transformed to (x, N, 2)
        start_note_values = T.alloc(np.array(0,dtype=np.int8), 1
, time_final.shape[1], 2 )
        correct_choices = self.output_mat[:,1:,0:-1,:].transpose
((2,0,1,3)).reshape((n_note-1,n_batch*n_time,2))
        note_choices_inputs = T.concatenate([start_note_values,
correct_choices], axis=0)

        # Together, this and the output from the last LSTM goes
to the new LSTM, but rotated, so that the batches in
        # one direction are the steps in the other, and vice ver
sa.
        note_inputs = T.concatenate( [time_final, note_choices_i
nputs], axis=2 )
        num_timebatch = note_inputs.shape[1]
```

```
        # apply dropout
        if self.dropout > 0:
            pitch_masks = theano_lstm.MultiDropout( [(num_timeba
tch, shape) for shape in self.p_layer_sizes], self.dropout)
        else:
            pitch_masks = []

        note_outputs_info = [initial_state_with_taps(layer, num_
timebatch) for layer in self.pitch_model.layers]
        note_result, _ = theano.scan(fn=step_note, sequences=[no
te_inputs], non_sequences=pitch_masks, outputs_info=note_outputs
_info)

        self.note_thoughts = note_result

        # Now note_result is a list of matrix [layer](note, batc
h/time, onOrArticProb) for each layer but we only care about
        # the hidden state of the last layer.
        # Transpose to be (batch, time, note, onOrArticProb)
        note_final = get_last_layer(note_result).reshape((n_note
,n_batch,n_time,2)).transpose(1,2,0,3)

        # The cost of the entire procedure is the negative log l
ikelihood of the events all happening.
        # For the purposes of training, if the ouputted probabil
ity is P, then the likelihood of seeing a 1 is P, and
        # the likelihood of seeing 0 is (1-P). So the likelihood
 is (1-P)(1-x) + Px = 2Px - P - x + 1
        # Since they are all binary decisions, and are all proba
bilities given all previous decisions, we can just
        # multiply the likelihoods, or, since we are logging the
m, add the logs.

        # Note that we mask out the articulations for those note
s that aren't played, because it doesn't matter
        # whether or not those are articulated.
        # The padright is there because self.output_mat[:,:,:,0]
 -> 3D matrix with (b,x,y), but we need 3d tensor with
        # (b,x,y,1) instead
        active_notes = T.shape_padright(self.output_mat[:,1:,:,0
])
        mask = T.concatenate([T.ones_like(active_notes),active_n
otes], axis=3)

        loglikelihoods = mask * T.log( 2*note_final*self.output_
mat[:,1:] - note_final - self.output_mat[:,1:] + 1 + self.epsilo
n )
        self.cost = T.neg(T.sum(loglikelihoods))

        updates, _, _, _, _ = create_optimization_updates(self.c
ost, self.params, method="adadelta")
        self.update_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs=self.cost,
            updates=updates,
```

```
            allow_input_downcast=True)

        self.update_thought_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs= ensure_list(self.time_thoughts) + ensure_li
st(self.note_thoughts) + [self.cost],
            allow_input_downcast=True)

    def _predict_step_note(self, in_data_from_time, *states):
        # States is [ *hiddens, last_note_choice ]
        hiddens = list(states[:-1])
        in_data_from_prev = states[-1]
        in_data = T.concatenate([in_data_from_time, in_data_from
_prev])

        # correct for dropout
        if self.dropout > 0:
            masks = [1 - self.dropout for layer in self.pitch_mo
del.layers]
            masks[0] = None
        else:
            masks = []

        new_states = self.pitch_model.forward(in_data, prev_hidd
ens=hiddens, dropout=masks)

        # Now new_states is a per-layer set of activations.
        probabilities = get_last_layer(new_states)

        # Thus, probabilities is a vector of two probabilities,
P(play), and P(artic | play)

        shouldPlay = self.srng.uniform() < (probabilities[0] **
self.conservativity)
        shouldArtic = shouldPlay * (self.srng.uniform() < probab
ilities[1])

        chosen = T.cast(T.stack(shouldPlay, shouldArtic), "int8"
)

        return ensure_list(new_states) + [chosen]

    def setup_predict(self):
        # In prediction mode, note steps are contained in the ti
me steps. So the passing gets a little bit hairy.

        self.predict_seed = T.bmatrix()
        self.steps_to_simulate = T.iscalar()

        def step_time(*states):
            # States is [ *hiddens, prev_result, time]
            hiddens = list(states[:-2])
            in_data = states[-2]
            time = states[-1]

            # correct for dropout
```

```
            if self.dropout > 0:
                masks = [1 - self.dropout for layer in self.time
_model.layers]
                masks[0] = None
            else:
                masks = []

            new_states = self.time_model.forward(in_data, prev_h
iddens=hiddens, dropout=masks)

            # Now new_states is a list of matrix [layer](notes,
hidden_states) for each layer
            time_final = get_last_layer(new_states)

            start_note_values = theano.tensor.alloc(np.array(0,d
type=np.int8), 2)

            # This gets a little bit complicated. In the trainin
g case, we can pass in a combination of the
            # time net's activations with the known choices. But
 in the prediction case, those choices don't
            # exist yet. So instead of iterating over the combin
ation, we iterate over only the activations,
            # and then combine in the previous outputs in the st
ep. And then since we are passing outputs to
            # previous inputs, we need an additional outputs_inf
o for the initial "previous" output of zero.
            note_outputs_info = ([ initial_state_with_taps(layer
) for layer in self.pitch_model.layers ] +
                                 [ dict(initial=start_note_value
s, taps=[-1]) ])

            notes_result, updates = theano.scan(fn=self._predict
_step_note, sequences=[time_final], outputs_info=note_outputs_in
fo)

            # Now notes_result is a list of matrix [layer/output
](notes, onOrArtic)
            output = get_last_layer(notes_result)

            next_input = OutputFormToInputFormOp()(output, time
+ 1) # TODO: Fix time
            #next_input = T.cast(T.alloc(0, 3, 4),'int64')

            return (ensure_list(new_states) + [ next_input, time
 + 1, output ]), updates

        # start_sentinel = startSentinel()
        num_notes = self.predict_seed.shape[0]

        time_outputs_info = ([ initial_state_with_taps(layer, nu
m_notes) for layer in self.time_model.layers ] +
                             [ dict(initial=self.predict_seed, t
aps=[-1]),
                               dict(initial=0, taps=[-1]),
                               None ])
```

```
        time_result, updates = theano.scan( fn=step_time,
                                            outputs_info=time_ou
tputs_info,
                                            n_steps=self.steps_t
o_simulate )

        self.predict_thoughts = time_result

        self.predicted_output = time_result[-1]

        self.predict_fun = theano.function(
            inputs=[self.steps_to_simulate, self.conservativity,
 self.predict_seed],
            outputs=self.predicted_output,
            updates=updates,
            allow_input_downcast=True)

        self.predict_thought_fun = theano.function(
            inputs=[self.steps_to_simulate, self.conservativity,
 self.predict_seed],
            outputs=ensure_list(self.predict_thoughts),
            updates=updates,
            allow_input_downcast=True)

    def setup_slow_walk(self):

        self.walk_input = theano.shared(np.ones((2,2), dtype='in
t8'))
        self.walk_time = theano.shared(np.array(0, dtype='int64'
))
        self.walk_hiddens = [theano.shared(np.ones((2,2), dtype=
theano.config.floatX)) for layer in self.time_model.layers if ha
s_hidden(layer)]

        # correct for dropout
        if self.dropout > 0:
            masks = [1 - self.dropout for layer in self.time_mod
el.layers]
            masks[0] = None
        else:
            masks = []

        new_states = self.time_model.forward(self.walk_input, pr
ev_hiddens=self.walk_hiddens, dropout=masks)

        # Now new_states is a list of matrix [layer](notes, hidd
en_states) for each layer
        time_final = get_last_layer(new_states)

        start_note_values = theano.tensor.alloc(np.array(0,dtype
=np.int8), 2)
        note_outputs_info = ([ initial_state_with_taps(layer) fo
r layer in self.pitch_model.layers ] +
                             [ dict(initial=start_note_values, t
aps=[-1]) ])
```

```
        notes_result, updates = theano.scan(fn=self._predict_ste
p_note, sequences=[time_final], outputs_info=note_outputs_info)

        # Now notes_result is a list of matrix [layer/output](no
tes, onOrArtic)
        output = get_last_layer(notes_result)

        next_input = OutputFormToInputFormOp()(output, self.walk
_time + 1) # TODO: Fix time
        #next_input = T.cast(T.alloc(0, 3, 4),'int64')

        slow_walk_results = (new_states[:-1] + notes_result[:-1]
 + [ next_input, output ])

        updates.update({
                self.walk_time: self.walk_time+1,
                self.walk_input: next_input
            })

        updates.update({hidden:newstate for hidden, newstate, la
yer in zip(self.walk_hiddens, new_states, self.time_model.layers
) if has_hidden(layer)})

        self.slow_walk_fun = theano.function(
            inputs=[self.conservativity],
            outputs=slow_walk_results,
            updates=updates,
            allow_input_downcast=True)

    def start_slow_walk(self, seed):
        seed = np.array(seed)
        num_notes = seed.shape[0]

        self.walk_time.set_value(0)
        self.walk_input.set_value(seed)
        for layer, hidden in zip((l for l in self.time_model.lay
ers if has_hidden(l)),self.walk_hiddens):
            hidden.set_value(np.repeat(np.reshape(layer.initial_
hidden_state.get_value(), (1,-1)), num_notes, axis=0))
```

```
sudo apt-get install -y gcc g++ gfortran build-essential git wge
```

```
t linux-image-generic libopenblas-dev python-dev python-pip pyth
on-nose python-numpy python-scipy
sudo pip install --upgrade --no-deps git+git://github.com/Theano
/Theano.git
sudo wget http://developer.download.nvidia.com/compute/cuda/repo
s/ubuntu1404/x86_64/cuda-repo-ubuntu1404_7.0-28_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1404_7.0-28_amd64.deb
sudo apt-get update
sudo apt-get install -y cuda
echo -e "\nexport PATH=/usr/local/cuda/bin:$PATH\n\nexport LD_LI
BRARY_PATH=/usr/local/cuda/lib64" >> .bashrc
sudo rebootsudo apt-get update
sudo apt-get -y dist-upgrade

cuda-install-samples-7.0.sh ~/
cd NVIDIA_CUDA-7.0_Samples/
cd 1_Utilities/deviceQuery
make
./deviceQuery

echo -e "\n[global]\nfloatX=float32\ndevice=gpu\nbase_compiledir
=~/external/.theano/\nallow_gc=False\nwarn_float64=warn\n[mode]=
FAST_RUN\n\n[nvcc]\nfastmath=True\n\n[cuda]\nroot=/usr/local/cud
a\n" >> ~/.theanorc
sudo pip install theano-lstm python-midi

sudo apt-get install htop reptyr

cd /usr/bin/
sudo wget https://raw.githubusercontent.com/aurora/rmate/master/
rmate
sudo chmod 775 rmate

# Mount new EBS volume (at sdf -> xvdf)
sudo fdisk /dev/xvdf
# Parameters:
# n
# p
# 1
#
#
# t
# 1
# 83
# w
sudo mkfs.ext3 -b 4096 /dev/xvdf

cd ~
mkdir external
sudo mount -t ext3 /dev/xvdf external/
sudo chmod 755 external
sudo chown ubuntu external
sudo chgrp ubuntu external
cd external
mkdir neural_music # and then get all files. Or maybe use git?
```

```
sudo dd if=/dev/zero of=~/external/swapfile1  bs=1024 count=4194
304
sudo chown root:root ~/external/swapfile1
sudo chmod 0600 ~/external/swapfile1
sudo mkswap ~/external/swapfile1
sudo swapon ~/external/swapfile1import itertools
from midi_to_statematrix import upperBound, lowerBound

def startSentinel():
    def noteSentinel(note):
        position = note
        part_position = [position]

        pitchclass = (note + lowerBound) % 12
        part_pitchclass = [int(i == pitchclass) for i in range(1
2)]

        return part_position + part_pitchclass + [0]*66 + [1]
    return [noteSentinel(note) for note in range(upperBound-lowe
rBound)]

def getOrDefault(l, i, d):
    try:
        return l[i]
    except IndexError:
        return d

def buildContext(state):
    context = [0]*12
    for note, notestate in enumerate(state):
        if notestate[0] == 1:
            pitchclass = (note + lowerBound) % 12
            context[pitchclass] += 1
    return context

def buildBeat(time):
    return [2*x-1 for x in [time%2, (time//2)%2, (time//4)%2, (t
ime//8)%2]]

def noteInputForm(note, state, context, beat):
    position = note
    part_position = [position]

    pitchclass = (note + lowerBound) % 12
    part_pitchclass = [int(i == pitchclass) for i in range(12)]
    # Concatenate the note states for the previous vicinity
    part_prev_vicinity = list(itertools.chain.from_iterable((get
OrDefault(state, note+i, [0,0]) for i in range(-12, 13))))

    part_context = context[pitchclass:] + context[:pitchclass]

    return part_position + part_pitchclass + part_prev_vicinity
+ part_context + beat + [0]

def noteStateSingleToInputForm(state,time):
    beat = buildBeat(time)
```

```
    context = buildContext(state)
    return [noteInputForm(note, state, context, beat) for note i
n range(len(state))]

def noteStateMatrixToInputForm(statematrix):
    # NOTE: May have to transpose this or transform it in some w
ay to make Theano like it
    #[startSentinel()] +
    inputform = [ noteStateSingleToInputForm(state,time) for tim
e,state in enumerate(statematrix) ]
    return inputform
```

Copyright (c) 2016, Daniel Johnson
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```
import midi, numpy

lowerBound = 24
upperBound = 102

def midiToNoteStateMatrix(midifile):
```

```
    pattern = midi.read_midifile(midifile)

    timeleft = [track[0].tick for track in pattern]

    posns = [0 for track in pattern]

    statematrix = []
    span = upperBound-lowerBound
    time = 0

    state = [[0,0] for x in range(span)]
    statematrix.append(state)
    while True:
        if time % (pattern.resolution / 4) == (pattern.resolutio
n / 8):
            # Crossed a note boundary. Create a new state, defau
lting to holding notes
            oldstate = state
            state = [[oldstate[x][0],0] for x in range(span)]
            statematrix.append(state)

        for i in range(len(timeleft)):
            while timeleft[i] == 0:
                track = pattern[i]
                pos = posns[i]

                evt = track[pos]
                if isinstance(evt, midi.NoteEvent):
                    if (evt.pitch < lowerBound) or (evt.pitch >=
 upperBound):
                        pass
                        # print "Note {} at time {} out of bound
s (ignoring)".format(evt.pitch, time)
                    else:
                        if isinstance(evt, midi.NoteOffEvent) or
 evt.velocity == 0:
                            state[evt.pitch-lowerBound] = [0, 0]
                        else:
                            state[evt.pitch-lowerBound] = [1, 1]
                elif isinstance(evt, midi.TimeSignatureEvent):
                    if evt.numerator not in (2, 4):
                        # We don't want to worry about non-4 tim
e signatures. Bail early!
                        # print "Found time signature event {}.
Bailing!".format(evt)
                        return statematrix

                try:
                    timeleft[i] = track[pos + 1].tick
                    posns[i] += 1
                except IndexError:
                    timeleft[i] = None

            if timeleft[i] is not None:
                timeleft[i] -= 1
```

```
        if all(t is None for t in timeleft):
            break

        time += 1

    return statematrix

def noteStateMatrixToMidi(statematrix, name="example"):
    statematrix = numpy.asarray(statematrix)
    pattern = midi.Pattern()
    track = midi.Track()
    pattern.append(track)

    span = upperBound-lowerBound
    tickscale = 55

    lastcmdtime = 0
    prevstate = [[0,0] for x in range(span)]
    for time, state in enumerate(statematrix + [prevstate[:]]):

        offNotes = []
        onNotes = []
        for i in range(span):
            n = state[i]
            p = prevstate[i]
            if p[0] == 1:
                if n[0] == 0:
                    offNotes.append(i)
                elif n[1] == 1:
                    offNotes.append(i)
                    onNotes.append(i)
            elif n[0] == 1:
                onNotes.append(i)
        for note in offNotes:
            track.append(midi.NoteOffEvent(tick=(time-lastcmdtim
e)*tickscale, pitch=note+lowerBound))
            lastcmdtime = time
        for note in onNotes:
            track.append(midi.NoteOnEvent(tick=(time-lastcmdtime
)*tickscale, velocity=40, pitch=note+lowerBound))
            lastcmdtime = time

        prevstate = state

    eot = midi.EndOfTrackEvent(tick=1)
    track.append(eot)

    midi.write_midifile("{}.mid".format(name), pattern)import os
, random
from midi_to_statematrix import *
from data import *
import pickle
#import cPickle as pickle

import signal
```

```
batch_width = 10 # number of sequences in a batch
batch_len = 16*8 # length of each sequence
division_len = 16 # interval between possible start locations

def loadPieces(dirpath):

    pieces = {}

    for fname in os.listdir(dirpath):
        if fname[-4:] not in ('.mid','.MID'):
            continue

        name = fname[:-4]

        outMatrix = midiToNoteStateMatrix(os.path.join(dirpath,
fname))
        if len(outMatrix) < batch_len:
            continue

        pieces[name] = outMatrix
        print ("Loaded {}".format(name))

    return pieces

def getPieceSegment(pieces):
    piece_output = random.choice(pieces.values())
    start = random.randrange(0,len(piece_output)-batch_len,divis
ion_len)
    # print "Range is {} {} {} -> {}".format(0,len(piece_output)
-batch_len,division_len, start)

    seg_out = piece_output[start:start+batch_len]
    seg_in = noteStateMatrixToInputForm(seg_out)

    return seg_in, seg_out

def getPieceBatch(pieces):
    i,o = zip(*[getPieceSegment(pieces) for _ in range(batch_wid
th)])
    return numpy.array(i), numpy.array(o)

def trainPiece(model,pieces,epochs,start=0):
    stopflag = [False]
    def signal_handler(signame, sf):
        stopflag[0] = True
    old_handler = signal.signal(signal.SIGINT, signal_handler)
    for i in range(start,start+epochs):
        if stopflag[0]:
            break
        error = model.update_fun(*getPieceBatch(pieces))
        if i % 100 == 0:
            print ("epoch {}, error={}".format(i,error))
        if i % 500 == 0 or (i % 100 == 0 and i < 1000):
            xIpt, xOpt = map(numpy.array, getPieceSegment(pieces
))
            noteStateMatrixToMidi(numpy.concatenate((numpy.expan
```

```
d_dims(xOpt[0], 0), model.predict_fun(batch_len, 1, xIpt[0])), a
xis=0),'output/sample{}'.format(i))
            pickle.dump(model.learned_config,open('output/params
{}.p'.format(i), 'wb'))
    signal.signal(signal.SIGINT, old_handler)
import theano, theano.tensor as T
import numpy as np

from data import noteStateSingleToInputForm


class OutputFormToInputFormOp(theano.Op):
    # Properties attribute
    __props__ = ()

    def make_node(self, state, time):
        state = T.as_tensor_variable(state)
        time = T.as_tensor_variable(time)
        return theano.Apply(self, [state, time], [T.bmatrix()])

    # Python implementation:
    def perform(self, node, inputs_storage, output_storage):
        state, time = inputs_storage
        output_storage[0][0] = np.array(noteStateSingleToInputFo
rm(state, time), dtype='int8')import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def actToColor(memcell, activation):
        return [0, sigmoid(activation), sigmoid(memcell)]

def internalMatrixToImgArray(inmat):
        return np.array(
                [[actToColor(m,a) for m,a in zip(row[:len(row)/2
],row[len(row)/2:])]
                        for row in inmat])

def probAndSuccessToImgArray(prob, succ, idx):
        return np.array([[[pr[idx]]*3,[sr[idx],0,0]] for pr, sr
in zip(prob, succ)])

def thoughtsToImageArray(thoughts):

        spacer = np.zeros((thoughts[0].shape[0], 5, 3))

        sequence = [
                        spacer,
                        probAndSuccessToImgArray(thoughts[4],tho
ughts[6], 0),
                        spacer,
                        probAndSuccessToImgArray(thoughts[4],tho
ughts[6], 1)
        ]
```

```
        for thought in thoughts[:-3]:
                sequence = [ spacer, internalMatrixToImgArray(th
ought) ] + sequence

        return (np.concatenate(sequence, axis=1 )*255).astype('u
int8')

def pastColor(prob, succ):
        return [prob[0], succ[0], succ[1]*succ[0]]

def drawPast(probs, succs):
        return np.array([
                [
                        pastColor(probs[time][note_idx], succs[t
ime][note_idx])
                        for time in range(len(probs))
                ]
                for note_idx in range(len(probs[0]))
        ])

def thoughtsAndPastToStackedArray(thoughts, probs, succs, len_pa
st):

        vert_spacer = np.zeros((thoughts[0].shape[0], 5, 3))

        past_out = drawPast(probs, succs)

        if len(probs) < len_past:
                past_out = np.pad(past_out, ((0,0),(len_past-len
(probs),0),(0,0)), mode='constant')

        def add_cur(ipt):
                return np.concatenate((
                        ipt,
                        vert_spacer,
                        probAndSuccessToImgArray(thoughts[-3],th
oughts[-1], 0),
                        vert_spacer,
                        probAndSuccessToImgArray(thoughts[-3],th
oughts[-1], 1)), axis=1)

        horiz_spacer = np.zeros((5, 1, 3))

        rows = [add_cur(past_out[-len_past:])]

        for thought in thoughts[:-3]:
                rows += [ horiz_spacer, add_cur(internalMatrixTo
ImgArray(thought)) ]

        maxlen = max([x.shape[1] for x in rows])
        rows = [np.pad(row, ((0,0),(maxlen-row.shape[1],0),(0,0)
), mode='constant') for row in rows]

        return (np.concatenate(rows, axis=0 )*255).astype('uint8
')
```