```
from __future__ import division
# different networks (autoencoder, conv autoencoder, recurrent)
# different signals (sine, recording)
# different noises (awgn, crowd)
# different domains (time, freq)
from numpy import complex64
import scipy
import lasagne
import theano
import theano.tensor as T
import numpy as np
from scikits.audiolab import wavwrite
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

SIMULATION_SNR = 6
FILE_SNR = '{} dB'.format(SIMULATION_SNR)
FILENAME_LOSS = 'plotfinal/curro-loss.csv'
FILENAME_MSE = 'plotfinal/curro-mse.csv'
LOSSFILE = open(FILENAME_LOSS, 'a')
MSEFILE = open(FILENAME_MSE, 'a')
LINEFMT = FILE_SNR + ',{}\n'
LINEFMTLOSS = FILE_SNR + ',{},{},{}\n'  # for dan net, we look at square loss &
 reg loss

LATENTFILE = open('plotfinal/dan-latent.csv', 'a')


dtype = theano.config.floatX
batchsize = 128
# framelen = 441
srate = 16000
pct = 0.5  # overlap
fftlen = 1024
framelen = fftlen
# overlap = int(framelen/2)

# dan-specific
shape = (batchsize,framelen)
latentsize = 2000
#background_latents_factor = 0.25
background_latents_factor = 0.5
minibatch_noise_only_factor = 0.5  # also for curro net
n_noise_only_examples = int(minibatch_noise_only_factor * batchsize)
n_background_latents = int(background_latents_factor * latentsize)
lambduh = 0.75

batch_norm = lasagne.layers.batch_norm

def mod_relu(x):
    eps = 1e-5
    return T.switch(x > eps, x, -eps/(x-1-eps))

def normalize(x):
    return x / max(abs(x))

def snr_after(x, x_hat):
    return np.var(x)/np.var(x-x_hat)
```

```
class ZeroOutBackgroundLatentsLayer(lasagne.layers.Layer):
    def __init__(self, incoming, **kwargs):
        super(ZeroOutBackgroundLatentsLayer, self).__init__(incoming)
        mask = np.ones((batchsize,latentsize))
        mask[:, 0:n_background_latents] = 0
        self.mask = theano.shared(mask, borrow=True)

    def get_output_for(self, input_data, reconstruct=False, **kwargs):
        if reconstruct:
            return self.mask * input_data
        return input_data


def dan_net():
    # net
    x = T.matrix('X')  # input
    y = T.matrix('Y')  # soft label
    network = batch_norm(lasagne.layers.InputLayer(shape, x))
    # network = lasagne.layers.InputLayer(shape, x)
    print network.output_shape
    network = lasagne.layers.DenseLayer(network, latentsize, nonlinearity=mo
d_relu)
    print network.output_shape
    latents = network
    network = ZeroOutBackgroundLatentsLayer(latents, background_latents_fact
or=background_latents_factor)
    network = lasagne.layers.DenseLayer(network, shape[1], nonlinearity=lasa
gne.nonlinearities.rectify)
    print network.output_shape

    # loss
    C = np.zeros((batchsize,latentsize))
    C[0:n_noise_only_examples, n_background_latents + 1:] = 1
    C_mat = theano.shared(np.asarray(C, dtype=dtype), borrow=True)
    mean_C = theano.shared(C.mean(), borrow=True)
    prediction = lasagne.layers.get_output(network)
    mse_term = lasagne.objectives.squared_error(prediction, x).sum(axis=[1],
 keepdims=True)
    scf = lambduh/mean_C
    regularization_term = scf * y * ((C_mat * lasagne.layers.get_output(late
nts))**2).sum(axis=[1], keepdims=True)
    loss = mse_term + regularization_term
    loss = loss.mean()

    # training compilation
    params = lasagne.layers.get_all_params(network, trainable=True)
    updates = lasagne.updates.adam(loss, params)
    train_fn = theano.function([x,y], loss, updates=updates)

    # inference compilation
    predict_fn = theano.function([x], lasagne.layers.get_output(network, det
erministic=True, reconstruct=True))

    #
    # other objectives
    #
```

```
    square_term = theano.function([x], mse_term.mean())
    regularization_term = theano.function([x,y], regularization_term.mean())

    def do_stuff(network, latents, predict_fn):
        pass

    latent_fn = theano.function([x], lasagne.layers.get_output(latents, dete
rministic=True))
    return network, latents, loss, square_term, regularization_term, train_f
n, predict_fn, do_stuff, latent_fn

def dan_main(params):
    network, latents, loss, square_loss, reg_loss, train_fn, predict_fn, do_
stuff, latent_fn = dan_net()
    lmse = []
    lsq = []
    lreg = []
    # inference example for simulations
    clean, noisy, n, labels = gen_freq_data(sample=True, gen_data_fn=gen_bat
ch_half_noisy_half_noise)

    for i in xrange(params.niter+1):
        _clean, _noisy, _n, _labels = gen_freq_data(sample=False, gen_data_f
n=gen_batch_half_noisy_half_noise)
        # swap 0 and 1 since for dan net, 0 is signal and 1 is background
        _labels = np.expand_dims(np.abs(_labels-1).astype(dtype)[:,1], axis=
1)
        # labels = np.abs(labels-1).astype(dtype)

        loss = train_fn(_noisy[0], _labels)
        lmse.append(loss)

        loss_lsq = square_loss(_noisy[0])
        lsq.append(loss_lsq)

        loss_reg = reg_loss(_noisy[0], _labels)
        lreg.append(loss_reg)
        print '%d\t%.3E\t%.3E\t%.3E' % (i, loss, loss_lsq, loss_reg)

        LOSSFILE.write(LINEFMTLOSS.format(loss, loss_lsq, loss_reg))

        if i in range(0, params.niter+50, 50):
            # validate mse
            cleaned_up = predict_fn(noisy[0])
            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
            baseline_mse = mean_squared_error(clean_time, noisy_time)
            print 'baseline mse:', baseline_mse
            mse = mean_squared_error(cleaned_up_time, clean_time)
            print 'mse:', mse
            MSEFILE.write(LINEFMT.format(mse))

            latentz = latent_fn(noisy[0])
            LATENTFILE.write('{},{}'.format(i, ','.join([str(x) for x in late
ntz])))

    cleaned_up = predict_fn(noisy[0])
```

```
    print 'freq mse:', mean_squared_error(cleaned_up, clean[0])
    cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
    cleaned_up_clean_phase = normalize(ISTFT(cleaned_up, clean[1], fftlen))
    clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
    noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
    print 'time mse noisy phase:', mean_squared_error(cleaned_up_time, clean_time)
    print 'time mse clean phase:', mean_squared_error(cleaned_up_clean_phase, clea
n_time)
    print 'baseline time mse noisy to clean:', mean_squared_error(noisy_time, clean_ti
me)
    wavwrite(clean_time, 'dan/x.wav', fs=srate, enc='pcm16')
    wavwrite(noisy_time, 'dan/y.wav', fs=srate, enc='pcm16')
    wavwrite(cleaned_up_time, 'dan/xhat.wav', fs=srate, enc='pcm16')
    wavwrite(cleaned_up_clean_phase, 'dan/xhat_cleanphase.wav', fs=srate, enc='pcm
16')

    plt.figure()
    plt.semilogy(lmse)
    plt.semilogy(lsq)
    plt.semilogy(lreg)
    plt.legend(['overall loss', 'squared error loss', 'regularization loss'])
    plt.savefig('dan/losses.svg', format='svg')

def paris_net(params):
    shape = (batchsize, fftlen)
    x = T.matrix('x')  # dirty
    s = T.matrix('s')  # clean
    #in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
    in_layer = lasagne.layers.InputLayer(shape, x)
    h1 = batch_norm(lasagne.layers.DenseLayer(in_layer, 2000, nonlinearity=m
od_relu))
    h1 = lasagne.layers.DenseLayer(h1, fftlen, nonlinearity=lasagne.nonlinea
rities.identity)

    # loss function
    prediction = lasagne.layers.get_output(h1)
    loss = lasagne.objectives.squared_error(prediction, s)
    return h1, x, s, loss.mean(), None, prediction

def curro_net(params):
    # input
    shape = (batchsize, framelen)
    x = T.matrix('x')  # dirty input
    label = T.matrix('label')  # noise OR signal/noise

    nonlin = mod_relu

    # network
    # in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))  # batch no
rm or no?
    in_layer = lasagne.layers.InputLayer(shape, x) # batch norm or no?
    layersizes = 1024*2
    h1 = lasagne.layers.DenseLayer(in_layer, layersizes, nonlinearity=nonlin
)
    h2 = lasagne.layers.DenseLayer(h1, layersizes, nonlinearity=nonlin)
    h3 = lasagne.layers.DenseLayer(h2, layersizes, nonlinearity=nonlin)
    f = h3  # at this point, first half is signal, second half is noise
```

```
    # signal split
    f_sig = lasagne.layers.SliceLayer(f, indices=slice(0,int(layersizes/2)),
 axis=-1)
    print 'sig split size: ', lasagne.layers.get_output_shape(f_sig)
    sig_d3 = lasagne.layers.DenseLayer(f_sig, framelen, nonlinearity=nonlin)
    # save parameters for noise split
    d3_W = sig_d3.W
    d3_b = sig_d3.b
    sig_d2 = lasagne.layers.DenseLayer(sig_d3, framelen, nonlinearity=nonlin
)
    d2_W = sig_d2.W
    d2_b = sig_d2.b
    g_sig = lasagne.layers.DenseLayer(sig_d2, framelen, nonlinearity=lasagne
.nonlinearities.identity)
    gs_W = g_sig.W
    gs_b = g_sig.b

    f_noi = lasagne.layers.SliceLayer(f, indices=slice(int(layersizes/2),lay
ersizes), axis=-1)
    print 'noisy split size: ', lasagne.layers.get_output_shape(f_noi)
    noi_d3 = lasagne.layers.DenseLayer(f_noi, framelen, W=d3_W, b=d3_b, nonl
inearity=nonlin)
    noi_d2 = lasagne.layers.DenseLayer(noi_d3, framelen, W=d2_W, b=d2_b, non
linearity=nonlin)
    g_noi = lasagne.layers.DenseLayer(noi_d2, framelen, W=gs_W, b=gs_b, nonl
inearity=lasagne.nonlinearities.identity)

    out_layer = lasagne.layers.ElemwiseSumLayer([g_sig,g_noi])

    prediction_sig = lasagne.layers.get_output(g_sig)
    prediction_noi = lasagne.layers.get_output(g_noi)
    # label is 1 for signal, 0 for noise
    prediction = label * prediction_sig + prediction_noi
    loss = lasagne.objectives.squared_error(prediction, x)
    loss_sig = lasagne.objectives.squared_error(prediction_sig, x)
    loss_noi = lasagne.objectives.squared_error(prediction_noi, x)

    return out_layer, g_sig, x, label, loss.mean(), g_noi, prediction, loss_
sig, loss_noi

def autoencoder(params):
    # network
    shape = (batchsize, framelen)
    x = T.matrix('x')  # dirty
    s = T.matrix('s')  # clean
    in_layer = batch_norm(lasagne.layers.InputLayer(shape, x))
    h1 = batch_norm(lasagne.layers.DenseLayer(in_layer, 400, nonlinearity=la
sagne.nonlinearities.leaky_rectify))
    h2 = batch_norm(lasagne.layers.DenseLayer(h1, 330, nonlinearity=lasagne.
nonlinearities.leaky_rectify))
    h3 = batch_norm(lasagne.layers.DenseLayer(h2, 300, nonlinearity=lasagne.
nonlinearities.leaky_rectify))
    h4 = batch_norm(lasagne.layers.DenseLayer(h3, 270, nonlinearity=lasagne.
nonlinearities.leaky_rectify))
    bottle = h4
    d4 = batch_norm(lasagne.layers.DenseLayer(h4, 300, nonlinearity=lasagne.
nonlinearities.leaky_rectify))
    d3 = batch_norm(lasagne.layers.DenseLayer(d4, 330, nonlinearity=lasagne.
```

```
nonlinearities.leaky_rectify))
    d2 = batch_norm(lasagne.layers.DenseLayer(d3, 400, nonlinearity=lasagne.
nonlinearities.leaky_rectify))
    x_hat = batch_norm(lasagne.layers.DenseLayer(d2, framelen, nonlinearity=
lasagne.nonlinearities.identity))

    # loss function
    prediction = lasagne.layers.get_output(x_hat)
    loss = lasagne.objectives.squared_error(prediction, s)
    reg = 2 * (1e-5 * lasagne.regularization.regularize_network_params(x_hat
, lasagne.regularization.l2) + \
          1e-6 * lasagne.regularization.regularize_network_params(x_hat, las
agne.regularization.l1))
    loss = loss + reg
    return x_hat, x, s, loss.mean(), reg.mean(), prediction

def train(autoencoder, x, s, loss):
    params = lasagne.layers.get_all_params(autoencoder, trainable=True)
    updates = lasagne.updates.adam(loss, params)
    train_fn = theano.function([x,s], loss, updates=updates)
    return train_fn

def gen_data(sample=False):
    def _sin_f(a, f, srate, n, phase):
        return a * np.sin(2*np.pi*f/srate*n+phase)

    def _noise_var(clean, snr_db):
        # we use one noise variance per minibatch
        avg_energy = np.sum(clean*clean)/clean.size
        snr_lin = 10**(snr_db/10)
        noise_var = avg_energy / snr_lin
        print '\tnoise variance for minibatch: ', noise_var
        return noise_var

    # f = 440
    if sample:
        n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)
        phase1 = np.random.uniform(0.0, 2*np.pi)
        phase2 = np.random.uniform(0.0, 2*np.pi)
        phase3 = np.random.uniform(0.0, 2*np.pi)
        phase4 = np.random.uniform(0.0, 2*np.pi)
        amp1 = np.random.uniform(0.25, 0.75)
        amp2 = np.random.uniform(0.25, 0.75)
        amp3 = np.random.uniform(0.25, 0.75)
        amp4 = np.random.uniform(0.25, 0.75)
    else:
        n = np.tile(np.linspace(0, framelen-1, framelen), (batchsize,1))
        phase1 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase2 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase3 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase4 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        amp1 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        amp2 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
```

```
)).transpose()
        amp3 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        amp4 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
    # clean = amp * np.sin(2 * np.pi * f / srate * n + phase)
    clean = _sin_f(amp1,441,srate,n,phase1) + \
            _sin_f(amp2,549,srate,n,phase2) + \
            _sin_f(amp3,660,srate,n,phase3) + \
            _sin_f(amp4,881,srate,n,phase4)

    # corrupt with gaussian noise
    var = _noise_var(clean, SIMULATION_SNR)
    noise = np.random.normal(0, var, clean.shape)
    noisy = clean + noise

    if sample:
        noisy = np.array([noisy[i:i+framelen] for i in xrange(0, len(noisy),
 int(pct*framelen))][0:batchsize])
        clean = np.array([clean[i:i+framelen] for i in xrange(0, len(clean),
 int(pct*framelen))][0:batchsize])
        #noisy = noisy.reshape(batchsize, framelen)
        #clean = clean.reshape(batchsize, framelen)

    return clean.astype(dtype), noisy.astype(dtype), n, None

def gen_batch_half_noisy_half_noise(sample=False):
    def _sin_f(a, f, srate, n, phase):
        return a * np.sin(2*np.pi*f/srate*n+phase)

    nop = minibatch_noise_only_factor  # noise only percentage of minibatch
    f = 440
    if sample:
        n = np.linspace(0, batchsize * framelen - 1, batchsize * framelen)
        np.random.seed(3)  # to get consistent samples
        phase1 = np.random.uniform(0.0, 2*np.pi)
        phase2 = np.random.uniform(0.0, 2*np.pi)
        phase3 = np.random.uniform(0.0, 2*np.pi)
        phase4 = np.random.uniform(0.0, 2*np.pi)
        amp1 = np.random.uniform(0.25, 0.75)
        amp2 = np.random.uniform(0.25, 0.75)
        amp3 = np.random.uniform(0.25, 0.75)
        amp4 = np.random.uniform(0.25, 0.75)
        np.random.seed()
        clean = _sin_f(amp1,441,srate,n,phase1) + \
                _sin_f(amp2,549,srate,n,phase2) + \
                _sin_f(amp3,660,srate,n,phase3) + \
                _sin_f(amp4,881,srate,n,phase4)
    else:
        n = np.tile(np.linspace(0, framelen-1, framelen), (batchsize,1))
        phase1 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase2 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase3 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
        phase4 = np.tile(np.random.uniform(0.0, 2*np.pi, batchsize), (framel
en, 1)).transpose()
```

```
        amp1 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        amp2 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        amp3 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        amp4 = np.tile(np.random.uniform(0.25, 0.75, batchsize), (framelen,1
)).transpose()
        # clean = amp * np.sin(2 * np.pi * f / srate * n + phase)
        clean = _sin_f(amp1,441,srate,n,phase1) + \
                _sin_f(amp2,549,srate,n,phase2) + \
                _sin_f(amp3,660,srate,n,phase3) + \
                _sin_f(amp4,881,srate,n,phase4)
        clean[0:int(batchsize*nop),:] = 0

    def _noise_var(clean, snr_db):
        # we use one noise variance per minibatch
        avg_energy = np.sum(clean*clean)/clean.size
        snr_lin = 10**(snr_db/10)
        noise_var = avg_energy / snr_lin
        print '\tnoise variance for minibatch: ', noise_var
        return noise_var

    # corrupt with gaussian noise
    # use only the signal examples do determine noise variance (in both case
s)
    if not sample:
        noise_var = _noise_var(clean[int(batchsize*nop):,:], SIMULATION_SNR)
    else:
        noise_var = _noise_var(clean[int(batchsize*nop):], SIMULATION_SNR)
    noise = np.random.normal(0, noise_var, clean.shape)
    noisy = clean + noise

    if sample:
        noisy = np.array([noisy[i:i+framelen] for i in xrange(0, len(noisy),
 int(pct*framelen))][0:batchsize])
        clean = np.array([clean[i:i+framelen] for i in xrange(0, len(clean),
 int(pct*framelen))][0:batchsize])

    if not sample:
        labels = np.ones((batchsize,1))
        labels[0:int(batchsize*nop)]=0
        # labels = np.zeros((batchsize,1))
        # labels[0:int(batchsize*nop)]=1
    else:
        # assuming "noisy" example for sample, not noise example
        labels = np.ones((batchsize,1))
        # labels = np.zeros((batchsize,1))
    labels = np.tile(labels, (1,framelen))

    return clean.astype(dtype), noisy.astype(dtype), n, labels.astype(dtype)

def stft(x, framelen, overlap=int(pct*framelen)):
    w = scipy.hanning(framelen)
    X = np.array([scipy.fft(w*x[i:i+framelen], freq_bins)
                    for i in range(0, len(x)-framelen, overlap)], dtype=com
plex64)
    X = np.transpose(X)
```

```
    return np.abs(X), np.angle(X)

def fft(x, fftlen):
    w = np.tile((scipy.hanning(fftlen)), (batchsize, 1))
    X = scipy.fft(w*x, fftlen, axis=-1)
    return np.abs(X).astype(dtype), np.angle(X).astype(dtype)

def gen_freq_data(sample=False, gen_data_fn=gen_data):
    # for training, use FFTs of any frames
    # for testing, use FFTs of frames with 25% overlap for proper reconstruc
tion
    clean, noisy, n, labels = gen_data_fn(sample)
    # get FFTs
    clean_stft = fft(clean, fftlen)  # mag, phase
    noisy_stft = fft(noisy, fftlen)  # mag, phase
    return clean_stft, noisy_stft, n, labels  # (mag, phase), (mag, phase)

def istft(X, framelen):
    frames_avg = int(1/pct)  # 4 in this case
    # no avg first,
    overlap = int(pct * framelen)
    #x = scipy.zeros(int(framelen/2*(time_bins + 1)))
    x = scipy.zeros(int(X.shape[1]*(X.shape[0]*pct+1-pct)))
    for n,i in enumerate(range(0, len(x)-framelen, overlap)):
        x[i:i+framelen] += scipy.real(scipy.ifft(X[n, :]))
    return x

def ISTFT(mag, phase, framelen):
    stft = mag * np.exp(1j*phase)
    # return np.fft.ifft(stft, framelen)
    return istft(stft, framelen)

def paris_main(params):
    a, x, s, loss, _, x_hat = paris_net({})
    train_fn = train(a,x,s,loss)
    lmse = []
    predict_fn = theano.function([x], x_hat)

    np.random.seed(3)
    clean, noisy, n, _ = gen_freq_data(sample=True)
    np.random.seed()

    for i in xrange(params.niter+1):
        _clean, _noisy, _n, _ = gen_freq_data()
        loss = train_fn(_noisy[0], _clean[0])
        LOSSFILE.write(LINEFMT.format(loss))
        lmse.append(loss)
        print i, loss

        if i in range(0,params.niter+50,50):
            # validate mse

            cleaned_up = predict_fn(noisy[0])
            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
            baseline_mse = mean_squared_error(clean_time, noisy_time)
            print 'baseline mse:', baseline_mse
```

```
            mse = mean_squared_error(cleaned_up_time, clean_time)
            print 'mse:', mse
            MSEFILE.write(LINEFMT.format(mse))

    clean, noisy, n, _ = gen_freq_data(sample=True)
    cleaned_up = predict_fn(noisy[0])
    cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
    clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
    mse = mean_squared_error(cleaned_up_time, clean_time)
    # print 'mse ', mse
    wavwrite(normalize(cleaned_up_time), 'paris/xhat.wav', fs=srate, enc='pcm16'
)
    wavwrite(normalize(clean_time), 'paris/x.wav', fs=srate, enc='pcm16')
    noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
    wavwrite(normalize(noisy_time), 'paris/n.wav', fs=srate, enc='pcm16')
    plt.figure()
    plt.subplot(411)
    #plt.plot(cleaned_up_time[0:fftlen*2])
    #plt.plot(clean_time[0:fftlen*2])
    plt.plot(cleaned_up_time[1000:1250])
    plt.plot(clean_time[1000:1250])
    plt.subplot(412)
    plt.semilogy(lmse)
    plt.subplot(413)
    plt.plot(clean[0][0,:])
    plt.subplot(414)
    plt.plot(np.unwrap(clean[1][0,:]))
    plt.savefig('paris/x.svg', format='svg')

def curro_main(params):
    g_sig, g_sig_for_real, x, s, loss, g_noi_for_real, x_hat, loss_sig, loss
_noi = curro_net({})
    train_fn = train(g_sig,x,s,loss)
    train_sig = theano.function([x], loss_sig.mean())
    train_noi = theano.function([x], loss_noi.mean())
    lmse = []
    lsig = []
    lnoi = []
    predict_fn = theano.function([x], lasagne.layers.get_output(g_sig_for_re
al, deterministic=True))
    predict_fn_noi = theano.function([x], lasagne.layers.get_output(g_noi_fo
r_real, deterministic=True))
    both = theano.function([x], lasagne.layers.get_output(g_sig, determinist
ic=True))

    np.random.seed(3)
    clean, noisy, n, labels = gen_freq_data(sample=True, gen_data_fn=gen_bat
ch_half_noisy_half_noise)
    np.random.seed()

    for i in xrange(params.niter+1):
        _clean, _noisy, _n, _labels = gen_freq_data(sample=False, gen_data_f
n=gen_batch_half_noisy_half_noise)
        loss = train_fn(_noisy[0], _labels)
        lmse.append(loss)

        loss1 = train_sig(_noisy[0])
        lsig.append(loss1)
```

```
        loss2 = train_noi(_noisy[0])
        lnoi.append(loss2)

        print i, loss, loss1, loss2
        LOSSFILE.write(LINEFMTLOSS.format(loss,loss1,loss2))

        if i in range(0,params.niter+50,50):
            # validate mse

            cleaned_up = predict_fn(noisy[0])
            cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
            clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
            noisy_time = normalize(ISTFT(noisy[0], noisy[1], fftlen))
            baseline_mse = mean_squared_error(clean_time, noisy_time)
            print 'baseline mse:', baseline_mse
            mse = mean_squared_error(cleaned_up_time, clean_time)
            print 'mse:', mse
            MSEFILE.write(LINEFMT.format(mse))


    cleaned_up = predict_fn(noisy[0])
    noisy_reconstructed = predict_fn_noi(noisy[0])
    both_ffts = both(noisy[0])

    cleaned_up_time = normalize(ISTFT(cleaned_up, noisy[1], fftlen))
    clean_time = normalize(ISTFT(clean[0], clean[1], fftlen))
    noisy_reconstructed = normalize(ISTFT(noisy_reconstructed, noisy[1], fft
len))
    both_time = normalize(ISTFT(both_ffts, noisy[1], fftlen))

    mse = mean_squared_error(cleaned_up_time, clean_time)
    mse_noi = mean_squared_error(noisy_reconstructed, clean_time)
    mse_both = mean_squared_error(both_time, clean_time)
    #print 'baseline mse', mean_squared_error()  TODO: mse
    print 'mse ', mse
    print 'mse of noisy half ', mse_noi
    print 'mse of combined (both) ', mse_both
    wavwrite(normalize(cleaned_up_time), 'curro/xhat.wav', fs=srate, enc='pcm16'
)
    wavwrite(normalize(clean_time), 'curro/x.wav', fs=srate, enc='pcm16')
    wavwrite(normalize(noisy_reconstructed), 'curro/nxhat.wav', fs=srate, enc='p
cm16')
    wavwrite(normalize(both_time), 'curro/both.wav', fs=srate, enc='pcm16')
    plt.figure()
    plt.subplot(511)
    plt.plot(clean_time[0:fftlen*3])
    plt.plot(cleaned_up_time[0:fftlen*3])
    plt.subplot(512)
    plt.semilogy(lmse)
    plt.subplot(513)
    #plt.plot(cleaned_up[0,:])
    plt.semilogy(np.abs(np.fft.fft(np.blackman(cleaned_up_time.size)*cleaned
_up_time)))
    plt.subplot(514)
    plt.plot(np.unwrap(noisy[1][0,:]))
    plt.subplot(515)
    plt.plot(noisy_reconstructed[0:fftlen*3])
```

```
    plt.savefig('curro/x.svg', format='svg')
    plt.figure()
    plt.plot(lsig)
    plt.plot(lnoi)
    plt.legend(['sig', 'noi'])
    plt.savefig('curro/split.svg', format='svg')

def sim_():
    # a, x, s, loss, reg, x_hat = autoencoder({})
    a, x, s, loss, _, x_hat = curro_net({})
    train_fn = train(a,x,s,loss)
    loss_mse = theano.function([x, s], loss)
    # loss_reg = theano.function([], reg)
    lmse = []
    # lreg = []
    predict_fn = theano.function([x,s], x_hat)
    # clean, noisy = gen_data()
    # wavwrite(clean[1,:], 'fig/s.wav', fs=srate, enc='pcm16')
    for i in xrange(niter):
        clean, noisy, _, labels = gen_freq_data(sample=False, gen_data_fn=ge
n_batch_half_noisy_half_noise)
        loss = train_fn(noisy, labels)
        lmse.append(loss)
        # lmse.append(loss_mse(noisy, clean))
        # lreg.append(loss_reg())
        print i, loss
    clean, noisy, n, labels = gen_batch_half_noisy_half_noise(sample=True)
    cleaned_up = predict_fn(noisy, labels)
    cleaned_up = cleaned_up.reshape(batchsize * framelen)
    # mse calculation
    mse = mean_squared_error(cleaned_up, clean.reshape(batchsize * framelen)
)
    print 'mse', mse
    wavwrite(clean.reshape(batchsize * framelen), 'fig/s.wav', fs=srate, enc='p
cm16')
    wavwrite(noisy.reshape(batchsize * framelen), 'fig/xn.wav', fs=srate, enc=
'pcm16')
    wavwrite(cleaned_up, 'fig/x.wav', fs=srate, enc='pcm16')
    plt.figure()
    plt.subplot(211)
    # plt.plot(n, clean.reshape(batchsize * framelen))
    # plt.plot(n, noisy.reshape(batchsize * framelen))
    # plt.plot(n, cleaned_up)
    plt.plot(n[0:framelen*2],clean[0:2,:].reshape(-1))
    plt.plot(n[0:framelen*2],noisy[0:2,:].reshape(-1))
    plt.plot(n[0:framelen*2],cleaned_up[0:framelen*2])
    # plt.plot(n[0:framelen],cleaned_up[0:framelen])
    plt.subplot(212)
    plt.plot(lmse)
    plt.semilogy(lmse)
    # plt.subplot(313)
    # plt.plot(lreg)
    # plt.semilogy(lreg)
    plt.savefig('fig/x.svg', format='svg')


if __name__ == "__main__":
    import sys
```

```
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('net', type=str, help='super, paris, dan, or curro', default='s
uper')
    parser.add_argument('-n', '--niter', type=int, help='number of iterations', defa
ult=2000)
    args = parser.parse_args()
    mapping = {
        'super': autoencoder,
        'paris': paris_main,
        'dan': dan_main,
        'curro': curro_main,
    }
    mapping[args.net](args)
    LOSSFILE.close()
    MSEFILE.close()
    LATENTFILE.close()
```