```python
"""
 This tutorial introduces stacked denoising auto-encoders (SdA) using Theano.

 Denoising autoencoders are the building blocks for SdA.
 They are based on auto-encoders as the ones used in Bengio et al. 2007.
 An autoencoder takes an input x and first maps it to a hidden representation
 y = f_{\theta}(x) = s(Wx+b), parameterized by \theta={W,b}. The resulting
 latent representation y is then mapped back to a "reconstructed" vector
 z \in [0,1]^d in input space z = g_{\theta'}(y) = s(W'y + b').  The weight
 matrix W' can optionally be constrained such that W' = W^T, in which case
 the autoencoder is said to have tied weights. The network is trained such
 that to minimize the reconstruction error (the error between x and z).

 For the denosing autoencoder, during training, first x is corrupted into
 \tilde{x}, where \tilde{x} is a partially destroyed version of x by means
 of a stochastic mapping. Afterwards y is computed as before (using
 \tilde{x}), y = s(W\tilde{x} + b) and z as s(W'y + b'). The reconstruction
 error is now measured between z and the uncorrupted input x, which is
 computed as the cross-entropy :
      - \sum_{k=1}^d[ x_k \log z_k + (1-x_k) \log( 1-z_k)]


 References :
   - P. Vincent, H. Larochelle, Y. Bengio, P.A. Manzagol: Extracting and
   Composing Robust Features with Denoising Autoencoders, ICML'08, 1096-1103,
   2008
   - Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle: Greedy Layer-Wise
   Training of Deep Networks, Advances in Neural Information Processing
   Systems 19, 2007

"""
import cPickle
import gzip
import os
import sys
import time

import numpy

import theano
import theano.tensor as T
from theano.tensor.shared_randomstreams import RandomStreams

from logistic_sgd import LogisticRegression, load_data
from mlp import HiddenLayer
from dA import dA
```

Look at what we are importing!
Everything we've learned so far

```python
class SdA(object):
    """Stacked denoising auto-encoder class (SdA)

    A stacked denoising autoencoder model is obtained by stacking several
    dAs. The hidden layer of the dA at layer `i` becomes the input of
    the dA at layer `i+1`. The first layer dA gets as input the input of
    the SdA, and the hidden layer of the last dA represents the output.
    Note that after pretraining, the SdA is dealt with as a normal MLP,
    the dAs are only used to initialize the weights.
    """

    def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
                 hidden_layers_sizes=[500, 500], n_outs=10,
                 corruption_levels=[0.1, 0.1]):
```

Construct the SdA

```python
""" This class is made to support a variable number of layers.

:type numpy_rng: numpy.random.RandomState
:param numpy_rng: numpy random number generator used to draw initial
            weights

:type theano_rng: theano.tensor.shared_randomstreams.RandomStreams
:param theano_rng: Theano random generator; if None is given one is
                   generated based on a seed drawn from `rng`

:type n_ins: int
:param n_ins: dimension of the input to the sdA

:type n_layers_sizes: list of ints
:param n_layers_sizes: intermediate layers size, must contain
                       at least one value

:type n_outs: int
:param n_outs: dimension of the output of the network

:type corruption_levels: list of float
:param corruption_levels: amount of corruption to use for each
                          layer
"""
```

stores the sigmoid layers

```python
self.sigmoid_layers = []
self.dA_layers = []
```

stores the denoising autoencoder layers

```python
self.params = []
self.n_layers = len(hidden_layers_sizes)

assert self.n_layers > 0

if not theano_rng:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
# allocate symbolic variables for the data
self.x = T.matrix('x')  # the data is presented as rasterized images
self.y = T.ivector('y')  # the labels are presented as 1D vector of
                         # [int] labels

# The SdA is an MLP, for which all weights of intermediate layers
# are shared with a different denoising autoencoders
# We will first construct the SdA as a deep multilayer perceptron,
# and when constructing each sigmoidal layer we also construct a
# denoising autoencoder that shares weights with that layer
# During pretraining we will train these autoencoders (which will
# lead to chainging the weights of the MLP as well)
# During finetunining we will finish training the SdA by doing
# stochastich gradient descent on the MLP
```

Loop over the hidden layers

```python
for i in xrange(self.n_layers):
    # construct the sigmoidal layer

    # the size of the input is either the number of hidden units of
    # the layer below or the input size if we are on the first layer
    if i == 0:
        input_size = n_ins
```

e.g. number of input pixels

```python
    else:
        input_size = hidden_layers_sizes[i - 1]
```

size of previous hidden layer

```python
    # the input to this layer is either the activation of the hidden
    # layer below or the input of the SdA if you are on the first
    # layer
```

e.g. first time, we pass in the pixels, second time we pass in the outputs from the

```
            if i == 0:
                layer_input = self.x
            else:
                layer_input = self.sigmoid_layers[-1].output
```

e.g. the raw input pixels

output from previous hidden layer

```
            sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                        input=layer_input,
                                        n_in=input_size,
                                        n_out=hidden_layers_sizes[i],
                                        activation=T.nnet.sigmoid)
            # add the layer to our list of layers
            self.sigmoid_layers.append(sigmoid_layer)
            # its arguably a philosophical question...
            # but we are going to only declare that the parameters of the
            # sigmoid_layers are parameters of the StackedDAA
            # the visible biases in the dA are parameters of those
            # dA, but not the SdA
            self.params.extend(sigmoid_layer.params)
```

Not exactly sure why it is a philosophical question?

```
            # Construct a denoising autoencoder that shared weights with this
            # layer
            dA_layer = dA(numpy_rng=numpy_rng,
                          theano_rng=theano_rng,
                          input=layer_input,
                          n_visible=input_size,
                          n_hidden=hidden_layers_sizes[i],
                          W=sigmoid_layer.W,
                          bhid=sigmoid_layer.b)
            self.dA_layers.append(dA_layer)
```

Input is the output from the previous SIGMOID layer

Create a denoising autoencoder layer

Weights and bias' are the same as the SIGMOID layer

We are out of the for loop over the layers

```
        # We now need to add a logistic layer on top of the MLP
        self.logLayer = LogisticRegression(
                         input=self.sigmoid_layers[-1].output,
                         n_in=hidden_layers_sizes[-1], n_out=n_outs)

        self.params.extend(self.logLayer.params)
        # construct a function that implements one step of finetunining

        # compute the cost for second phase of training,
        # defined as the negative log likelihood
        self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
        # compute the gradients with respect to the model parameters
        # symbolic variable that points to the number of errors made on the
        # minibatch given by self.x and self.y
        self.errors = self.logLayer.errors(self.y)
```

Create the Logistic Regression layer

input is the output of the last SIGMOID layer, output is the class labels

like in the MLP

this is the architecture, skip down to look at the creating the SdA

```
    def pretraining_functions(self, train_set_x, batch_size):
        ''' Generates a list of functions, each of them implementing one
        step in trainnig the dA corresponding to the layer with same index.
        The function will require as input the minibatch index, and to train
        a dA you just need to iterate, calling the corresponding function on
        all minibatch indexes.

        :type train_set_x: theano.tensor.TensorType
        :param train_set_x: Shared variable that contains all datapoints used
                            for training the dA

        :type batch_size: int
        :param batch_size: size of a [mini]batch

        :type learning_rate: float
        :param learning_rate: learning rate used during training for any of
```

For each layer, get a list of functions that updates the weights

```
                             the dA layers
        '''

        # index to a [mini]batch
        index = T.lscalar('index')  # index to a minibatch
        corruption_level = T.scalar('corruption')  # % of corruption to use
        learning_rate = T.scalar('lr')  # learning rate to use
        # number of batches
        n_batches = train_set_x.get_value(borrow=True).shape[0] / batch_size
        # begining of a batch, given `index`
        batch_begin = index * batch_size
        # ending of a batch given `index`
        batch_end = batch_begin + batch_size
```

these are parameters to the function...

mini-batch stuff

```
        pretrain_fns = []
        for dA in self.dA_layers:
```

for each layer in the dA

```
            # get the cost and the updates list
            cost, updates = dA.get_cost_updates(corruption_level,
                                                learning_rate)
```

get the cost to reconstruct the input, and update the weights

```
            # compile the theano function
            fn = theano.function(inputs=[index,
                              theano.Param(corruption_level, default=0.2),
                              theano.Param(learning_rate, default=0.1)],
                                 outputs=cost,
                                 updates=updates,
                                 givens={self.x: train_set_x[batch_begin:
                                                             batch_end]})
```

a function that will compute the cost and the weight updates

```
            # append `fn` to the list of functions
            pretrain_fns.append(fn)

        return pretrain_fns
```

list of compiled pre-training functions used to train each layer, where we can index the layer by *i*

```
    def build_finetune_functions(self, datasets, batch_size, learning_rate):
        '''Generates a function `train` that implements one step of
        finetuning, a function `validate` that computes the error on
        a batch from the validation set, and a function `test` that
        computes the error on a batch from the testing set

        :type datasets: list of pairs of theano.tensor.TensorType
        :param datasets: It is a list that contain all the datasets;
                         the has to contain three pairs, `train`,
                         `valid`, `test` in this order, where each pair
                         is formed of two Theano variables, one for the
                         datapoints, the other for the labels

        :type batch_size: int
        :param batch_size: size of a minibatch

        :type learning_rate: float
        :param learning_rate: learning rate used during finetune stage
        '''

        (train_set_x, train_set_y) = datasets[0]
        (valid_set_x, valid_set_y) = datasets[1]
        (test_set_x, test_set_y) = datasets[2]

        # compute number of minibatches for training, validation and testing
        n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
        n_valid_batches /= batch_size
        n_test_batches = test_set_x.get_value(borrow=True).shape[0]
        n_test_batches /= batch_size
```

Supervised fine-tuning

```
index = T.lscalar('index')  # index to a [mini]batch

# compute the gradients with respect to the model parameters
gparams = T.grad(self.finetune_cost, self.params)

# compute list of fine-tuning updates
updates = []
for param, gparam in zip(self.params, gparams):
    updates.append((param, param - gparam * learning_rate))

train_fn = theano.function(inputs=[index],
      outputs=self.finetune_cost,
      updates=updates,
      givens={
        self.x: train_set_x[index * batch_size:
                            (index + 1) * batch_size],
        self.y: train_set_y[index * batch_size:
                            (index + 1) * batch_size]},
      name='train')

test_score_i = theano.function([index], self.errors,
         givens={
           self.x: test_set_x[index * batch_size:
                              (index + 1) * batch_size],
           self.y: test_set_y[index * batch_size:
                              (index + 1) * batch_size]},
              name='test')

valid_score_i = theano.function([index], self.errors,
      givens={
         self.x: valid_set_x[index * batch_size:
                             (index + 1) * batch_size],
         self.y: valid_set_y[index * batch_size:
                             (index + 1) * batch_size]},
              name='valid')

# Create a function that scans the entire validation set
def valid_score():
    return [valid_score_i(i) for i in xrange(n_valid_batches)]

# Create a function that scans the entire test set
def test_score():
    return [test_score_i(i) for i in xrange(n_test_batches)]

return train_fn, valid_score, test_score
```

Gradients computed like in MLP

Updates to the weights

Divide data into train, validate and test with the features and corresponding labels

Get the validation and test scores, returns the error

```
def test_SdA(finetune_lr=0.1, pretraining_epochs=15,
             pretrain_lr=0.001, training_epochs=1000,
             dataset='mnist.pkl.gz', batch_size=1):
    """
    Demonstrates how to train and test a stochastic denoising autoencoder.

    This is demonstrated on MNIST.

    :type learning_rate: float
    :param learning_rate: learning rate used in the finetune stage
    (factor for the stochastic gradient)

    :type pretraining_epochs: int
    :param pretraining_epochs: number of epoch to do pretraining
```

Here's the test code for the SdA

```
    :type pretrain_lr: float
    :param pretrain_lr: learning rate to be used during pre-training

    :type n_iter: int
    :param n_iter: maximal number of iterations ot run the optimizer

    :type dataset: string
    :param dataset: path the the pickled dataset

    """

    datasets = load_data(dataset)

    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]
```

Load the data, split into train, validate and test set

```
    # compute number of minibatches for training, validation and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0]
    n_train_batches /= batch_size

    # numpy random generator
    numpy_rng = numpy.random.RandomState(89677)
    print '... building the model'
    # construct the stacked denoising autoencoder class
    sda = SdA(numpy_rng=numpy_rng, n_ins=28 * 28,
              hidden_layers_sizes=[1000, 1000, 1000],
              n_outs=10)
```

Here we actually construct the stacked autoencoder

```
    #########################
    # PRETRAINING THE MODEL #
    #########################
    print '... getting the pretraining functions'
    pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
                                                batch_size=batch_size)
```

Pretraining functions for the **autoencoder** layers to learn to reconstruct the input

```
    print '... pre-training the model'
    start_time = time.clock()
    ## Pre-train layer-wise
    corruption_levels = [.1, .2, .3]
    for i in xrange(sda.n_layers):
        # go through pretraining epochs
        for epoch in xrange(pretraining_epochs):
            # go through the training set
            c = []
            for batch_index in xrange(n_train_batches):
                c.append(pretraining_fns[i](index=batch_index,
                         corruption=corruption_levels[i],
                         lr=pretrain_lr))
            print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),
            print numpy.mean(c)
```

Corrupt it more over the 3 layers

For each of the layers...

For 1000 epochs...

For all the mini-batches...

update the weights in layer *i*

user-specified learning rate

average error for the epoch over the layer

```
    end_time = time.clock()

    print >> sys.stderr, ('The pretraining code for file ' +
                          os.path.split(__file__)[1] +
                          ' ran for %.2fm' % ((end_time - start_time) / 60.))

    ########################
    # FINETUNING THE MODEL #
    ########################
```

```
# get the training, validation and testing function for the model
print '... getting the finetuning functions'
train_fn, validate_model, test_model = sda.build_finetune_functions(
            datasets=datasets, batch_size=batch_size,
            learning_rate=finetune_lr)
```

Functions to do the supervised fine-tuning treating this like a MLP

```
print '... finetunning the model'
# early-stopping parameters
patience = 10 * n_train_batches  # look as this many examples regardless
patience_increase = 2.  # wait this much longer when a new best is
                        # found
improvement_threshold = 0.995  # a relative improvement of this much is
                               # considered significant
validation_frequency = min(n_train_batches, patience / 2)
                              # go through this many
                              # minibatche before checking the network
                              # on the validation set; in this case we
                              # check every epoch

best_params = None
```

where is this used?

```
best_validation_loss = numpy.inf
test_score = 0.
start_time = time.clock()
```

keep track of the best scores

```
done_looping = False
epoch = 0
```

Can terminate earlier

```
while (epoch < training_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in xrange(n_train_batches):
```

Loop over the mini-batches

```
        minibatch_avg_cost = train_fn(minibatch_index)
```

Call the supervised training function and get the error

```
        iter = (epoch - 1) * n_train_batches + minibatch_index

        if (iter + 1) % validation_frequency == 0:
            validation_losses = validate_model()
```

Get the errors over the validation set

```
            this_validation_loss = numpy.mean(validation_losses)
            print('epoch %i, minibatch %i/%i, validation error %f %%' %
                  (epoch, minibatch_index + 1, n_train_batches,
                   this_validation_loss * 100.))

            # if we got the best validation score until now
            if this_validation_loss < best_validation_loss:

                #improve patience if loss improvement is good enough
                if (this_validation_loss < best_validation_loss *
                    improvement_threshold):
                    patience = max(patience, iter * patience_increase)
```

Think this means we keep training if the results are somewhat improving

```
                # save best validation score and iteration number
                best_validation_loss = this_validation_loss
                best_iter = iter

                # test it on the test set
                test_losses = test_model()
                test_score = numpy.mean(test_losses)
                print(('     epoch %i, minibatch %i/%i, test error of '
                       'best model %f %%') %
                      (epoch, minibatch_index + 1, n_train_batches,
                       test_score * 100.))

        if patience <= iter:
            done_looping = True
```

```
                break

    end_time = time.clock()
    print(('Optimization complete with best validation score of %f %%,'
           'with test performance %f %%') %
                 (best_validation_loss * 100., test_score * 100.))
    print >> sys.stderr, ('The training code for file ' +
                          os.path.split(__file__)[1] +
                          ' ran for %.2fm' % ((end_time - start_time) / 60.))


if __name__ == '__main__':
    test_SdA()
```