In [1]:
```
%load_ext autoreload
%autoreload 2
%pylab inline
%aimport numpy
np=numpy
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
import numpy
import theano
import theano.tensor as T

class MLP(object):
    """multilayer perceptron"""
    def __init__(self, rng, input, n_input, n_hidden, n_output):

        # hidden layer weights
        self.W1 = theano.shared(
        numpy.asarray(
            rng.uniform(
                        low=-numpy.sqrt(1. / n_input),
                        high=numpy.sqrt(1. / n_input),
                        size=(n_input, n_hidden)),
            dtype=theano.config.floatX),
            name='W1', borrow=True)

        # hidden layer biases
        self.b1 = theano.shared(numpy.asarray(numpy.zeros(n_hidden,),
                                dtype=theano.config.floatX),
                                name='b1', borrow=True)

        # output layer weights
        self.W2 = theano.shared(
        numpy.asarray(
            rng.uniform(
                        low=-numpy.sqrt(1. / n_input),
                        high=numpy.sqrt(1. / n_input),
                        size=(n_hidden, n_output)),
            dtype=theano.config.floatX),
            name='W2', borrow=True)

        # output layer biases
        self.b2 = theano.shared(numpy.asarray(numpy.zeros(n_output,),
                                dtype=theano.config.floatX),
                                name='b2', borrow=True)

        # prediction formula
        self.p_y_given_x = T.nnet.softmax(T.dot(T.tanh(T.dot(input, se
lf.W1) + self.b1), self.W2) + self.b2)

        # predicted y from x
        self.y_pred = T.argmax(self.p_y_given_x, axis=1)
```

```
        # group all the parameters of the model
        self.params = [self.W1, self.b1, self.W2, self.b2]

        # the penalty functions
        self.L1 = abs(self.W1).sum() + abs(self.W2).sum()
        self.L2 = (self.W1**2).sum() + (self.W2**2).sum()



    def negative_log_likelihood(self, y):
        return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y
])

    def errors(self, y):
        # check if y has same dimension of y_pred
        if y.ndim != self.y_pred.ndim:
            raise TypeError(
                'y should have the same shape as self.y_pred',
                ('y', y.type, 'y_pred', self.y_pred.type)
            )
        # check if y is of the correct datatype
        if y.dtype.startswith('int'):
            # the T.neq operator returns a vector of 0s and 1s, where
1
            # represents a mistake in prediction
            return T.mean(T.neq(self.y_pred, y))
        else:
            raise NotImplementedError()
```

In [3]:
```
def load_data():
    '''
    Loads the data
    '''

    test_x = numpy.loadtxt('test_images.txt', delimiter=',')
    test_y = numpy.argmax(numpy.loadtxt('test_labels.txt', delimiter='
,'), axis=1)
    train_x = numpy.loadtxt('train_images.txt', delimiter=',')
    train_y = numpy.argmax(numpy.loadtxt('train_labels.txt', delimiter
=','), axis=1)


    def shared_dataset(data_x, data_y, borrow=True):
        """
        Function that loads the dataset into shared variables

        The reason we store our dataset in shared variables is to allo
w
        Theano to copy it into the GPU memory (when code is run on GPU
).
        Since copying data into the GPU is slow, copying a minibatch e
verytime
        is needed (the default behaviour if the data is not in a share
d

        variable) would lead to a large decrease in performance.
        """
        shared_x = theano.shared(numpy.asarray(data_x,
                                               dtype=theano.config.flo
atX),
                                 borrow=borrow)
        shared_y = theano.shared(numpy.asarray(data_y,
                                               dtype=theano.config.flo
atX),
                                 borrow=borrow)
        # When storing data on the GPU it has to be stored as floats
        # therefore we will store the labels as ``floatX`` as well
        # (``shared_y`` does exactly that). But during our computation
s
        # we need them as ints (we use labels as index, and if they ar
e
        # floats it doesn't make sense) therefore instead of returning
        # ``shared_y`` we will have to cast it to int. This little hac
k
        # lets ous get around this issue
        return shared_x, T.cast(shared_y, 'int32')

    test_x, test_y = shared_dataset(test_x, test_y)
    train_x, train_y = shared_dataset(train_x, train_y)

    rval = (train_x, train_y, test_x, test_y)
    return rval
```

In [4]:
```
def run_test(n_epochs,
```

```
              n_hidden,
              learning_rate=0.01,
              L1_reg=0.00,
              L2_reg=0.0001,
              batch_size=100,
              check_gradients=False):
    n_input = 28*28
    n_output =10

    # only check the fist batch if debug
    if check_gradients==True:
        batch_size=1

    # load the data
    datasets = load_data()
    train_set_x, train_set_y, test_set_x, test_set_y = datasets

    # compute number of minibatches for training and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0] / ba
tch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] / batc
h_size

    print '\n... building the model'

    # allocate symbolic variables for the data
    index = T.lscalar()  # index to a minibatch
    x = T.matrix('x')    # the data is presented as rasterized images
    y = T.ivector('y')   # the labels are presented as 1D vector of in
t labels

    rng = numpy.random.RandomState(1234)

    # construct the MLP class
    classifier = MLP(rng, x, n_input, n_hidden, n_output)

    # minimize negative log likelihood of the model
    # and the regularization terms (L1 and L2)
    # cost is expressed symbolically
    cost = (
            classifier.negative_log_likelihood(y)
            + L1_reg * classifier.L1
            + L2_reg * classifier.L2
            )

    # returns the cost
    ret_cost = theano.function(
    inputs=[index],
    outputs=cost,
    givens={
            x: train_set_x[index * batch_size:(index + 1) * batch_size
],
            y: train_set_y[index * batch_size:(index + 1) * batch_size
]
            }
        )
```

```
    # fit on train set
    check_fit_train_set = theano.function(
        inputs=[],
        outputs=classifier.errors(y),
        givens={
                x: train_set_x,
                y: train_set_y
                }
            )

    # fit on test set
    check_fit_test_set = theano.function(
        inputs=[],
        outputs=classifier.errors(y),
        givens={
                x: test_set_x,
                y: test_set_y
                }
            )

    # compile a Theano function that computes the mistakes that are ma
de
    # by the model on a minibatch of the train set (we'll see overfitt
ing)
    check_fit_batch = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
            x: train_set_x[index * batch_size:(index + 1) * batch_size
],
            y: train_set_y[index * batch_size:(index + 1) * batch_size
]
            }
        )

    # compute the gradient of cost with respect to theta (sorted in pa
rams)
    # the resulting gradients will be stored in a list gparams
    gradient_w1 = T.grad(cost, classifier.W1)
    gradient_b1 = T.grad(cost, classifier.b1)

    gradient_w2 = T.grad(cost, classifier.W2)
    gradient_b2 = T.grad(cost, classifier.b2)

    # specify how to update the parameters of the model as a list of
    # (variable, update expression) pairs
    updates = [
        (param, param - learning_rate * gradient)
        for param, gradient in [(classifier.W1, gradient_w1),
                                (classifier.b1, gradient_b1),
                                (classifier.W2, gradient_w2),
                                (classifier.b2, gradient_b2)]]

    values = [gradient_w1, gradient_b1, gradient_w2, gradient_b2]

    ret_gradient = theano.function(
```

```
        inputs=[index],
        outputs=values,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_siz
e],
            y: train_set_y[index * batch_size: (index + 1) * batch_siz
e]
        }
    )

    # updates the parameter of the model based on the rules defined in
 'updates'
    train_model = theano.function(
        inputs=[index],
        outputs=[],
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_siz
e],
            y: train_set_y[index * batch_size: (index + 1) * batch_siz
e]
        }
    )

    print '... training'

    best_iter = 0
    test_scores = []
    train_scores = []

    epoch = 0

    while (epoch < n_epochs):

        epoch = epoch + 1

        for minibatch_index in xrange(n_train_batches):
            train_model(minibatch_index)
            if check_gradients==True:
                symbolic_gradients = ret_gradient(minibatch_index)

                epsilon = 1E-4
                a = numpy.zeros((n_input, n_hidden))
                b = numpy.zeros((n_hidden,))
                c = numpy.zeros((n_hidden, n_output))
                d = numpy.zeros((n_output,))
                grad=  [a,b,c,d]

                # b1
                b1_vals = classifier.b1.get_value()
                for i in range(len(grad[1])):
                    # compute low
                    b1_vals[i] -= epsilon
                    classifier.b1.set_value(b1_vals, borrow=True)
                    low= ret_cost(minibatch_index)
```

```
            # compute high
            b1_vals[i] += 2.*epsilon
            classifier.b1.set_value(b1_vals, borrow=True)
            high= ret_cost(minibatch_index)

            # reset the value
            b1_vals[i] -= epsilon
            # store the gradient
            grad[1][i] = (high - low) / (2.*epsilon)

        # b2
        b2_vals = classifier.b2.get_value()
        for i in range(len(grad[3])):
            # compute low
            b2_vals[i] -= epsilon
            classifier.b2.set_value(b2_vals, borrow=True)
            low= ret_cost(minibatch_index)

            # compute high
            b2_vals[i] += 2.*epsilon
            classifier.b2.set_value(b2_vals, borrow=True)
            high= ret_cost(minibatch_index)

            # reset the value
            b2_vals[i] -= epsilon

            # store the gradient
            grad[3][i] = (high - low) / (2.*epsilon)


        # W1
        w1_vals = classifier.W1.get_value()
        for i in range(len(grad[0])):
            for j in range(len(grad[0][0])):

                # compute low
                w1_vals[i][j] -= epsilon
                classifier.W1.set_value(w1_vals, borrow=True)
                low= ret_cost(minibatch_index)

                # compute high
                w1_vals[i][j] += 2.*epsilon
                classifier.W1.set_value(w1_vals, borrow=True)
                high= ret_cost(minibatch_index)

                # reset the value
                w1_vals[i][j] -= epsilon

                # store the gradient
                grad[0][i][j] = (high - low) / (2.*epsilon)

        # W2
        w2_vals = classifier.W2.get_value()
        for i in range(len(grad[2])):
            for j in range(len(grad[2][0])):
```

```
                        # compute low
                        w2_vals[i][j] -= epsilon
                        classifier.W2.set_value(w2_vals, borrow=True)
                        low= ret_cost(minibatch_index)

                        # compute high
                        w2_vals[i][j] += 2.*epsilon
                        classifier.W2.set_value(w2_vals, borrow=True)
                        high= ret_cost(minibatch_index)

                        # reset the value
                        w2_vals[i][j] -= epsilon

                        # store the gradient
                        grad[2][i][j] = (high - low) / (2.*epsilon)

                return (symbolic_gradients, grad)

        # verify the fit on the datasets
        train_scores.append(check_fit_train_set())
        test_scores.append(check_fit_test_set())


    print '... done\n'
    return (train_scores, test_scores)
```

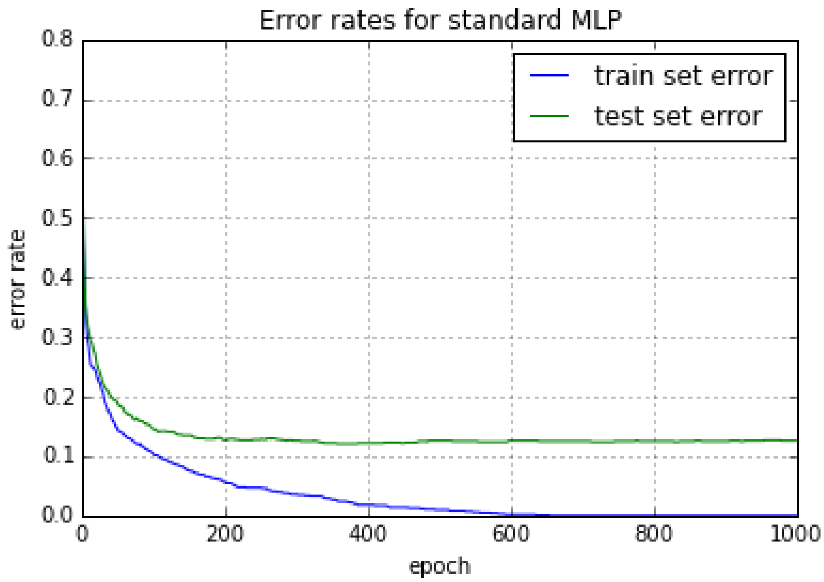
In [5]:

```
train_scores, test_scores = run_test(1000, 500)
```

```
... building the model
... training
... done
```

In [6]:

```
plt.plot(train_scores, label='train set error')
plt.plot(test_scores, label='test set error')
plt.xlabel('epoch')
plt.ylabel('error rate')
plt.grid(True)
plt.legend()
plt.title('Error rates for standard MLP')
plt.show()
print "Parameters are the following: epochs=1000, n_hidden=500, learni
ng_rate=0.01, L1_reg=0.00, L2_reg=0.0001, batch_size=100"
```

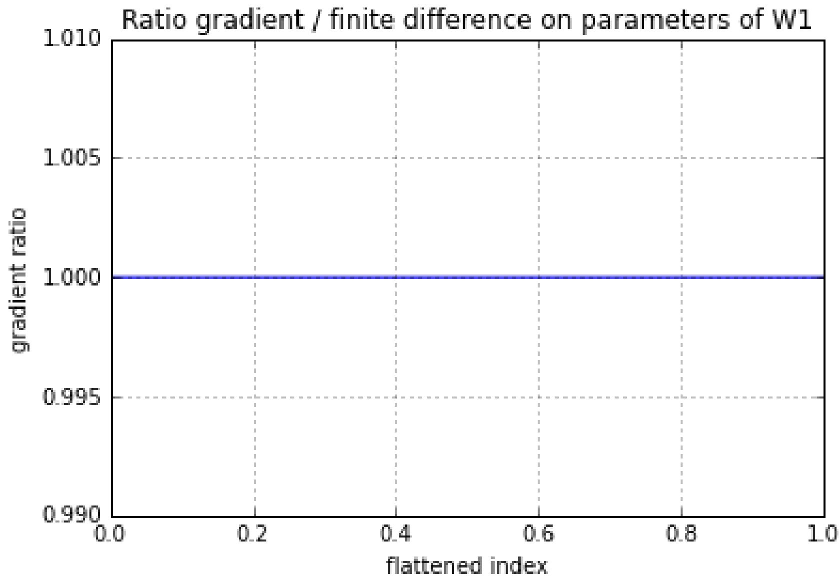


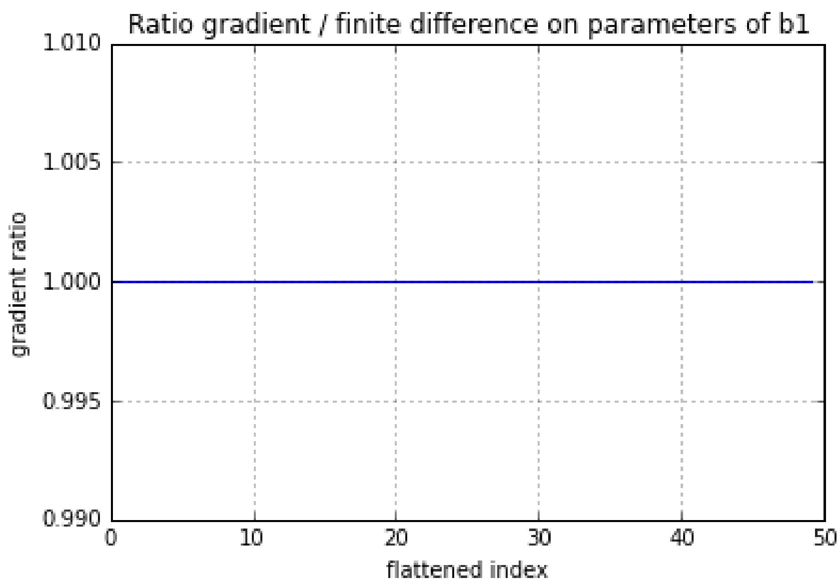

```
Parameters are the following: epochs=1000, n_hidden=500, learning_rate
=0.01, L1_reg=0.00, L2_reg=0.0001, batch_size=100
```

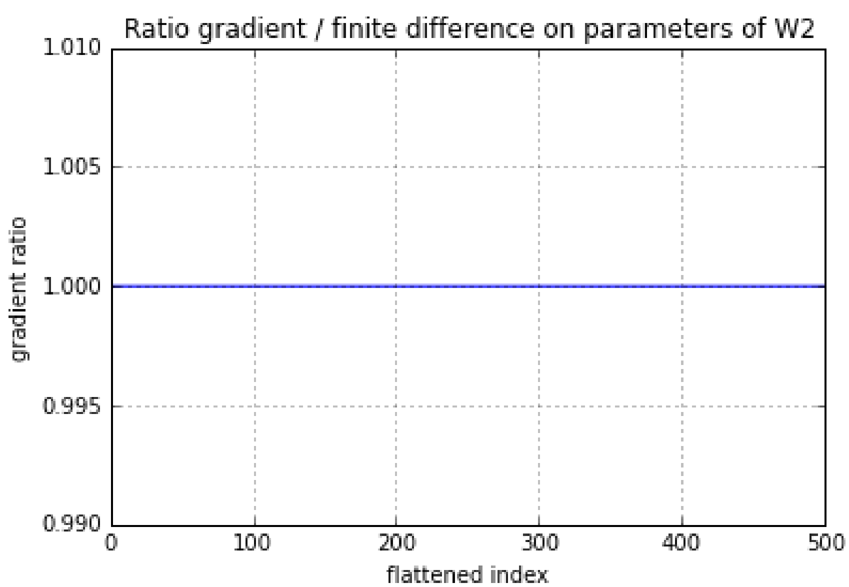
In [8]:

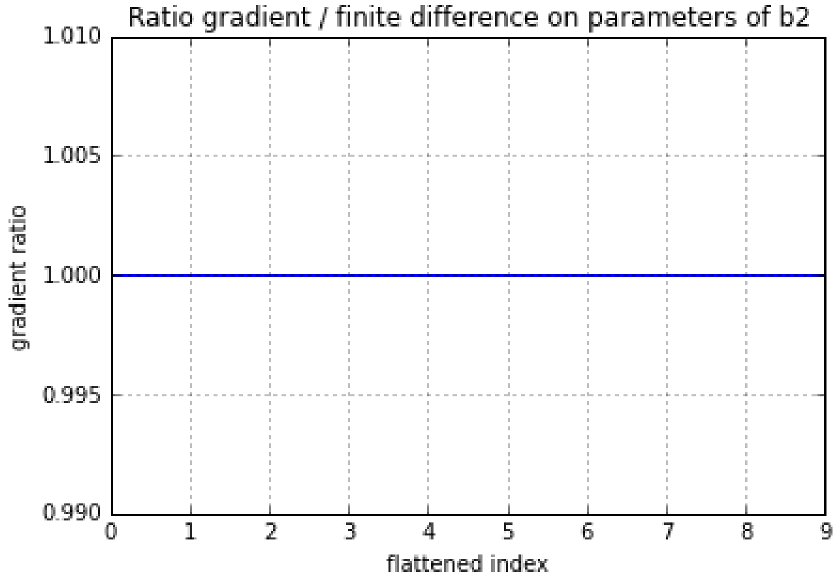
```
import matplotlib.pyplot as plt

theano_grad, finite_diff_grad = run_test(1000, 50, check_gradients=Tru
e)

pylab.gca().set_autoscale_on(False)

theano_grad_flat = map(numpy.ndarray.flatten, theano_grad)
finite_diff_grad_flat = map(numpy.ndarray.flatten, finite_diff_grad)

labels = ['W1', 'b1', 'W2', 'b2']
for i in range(len(theano_grad_flat)):
    plt.figure(i+1)
    data = theano_grad_flat[i] / finite_diff_grad_flat[i]
    plt.plot(data)
    plt.xlabel('flattened index')
    plt.ylabel('gradient ratio')
    plt.grid(True)
    plt.title('Ratio gradient / finite difference on parameters of {0}
'.format(labels[i]))
    ylim((0.99, 1.01))
```

```
... building the model
... training
```









In [9]:
```
class MLP_relu(MLP):
    # modified with max(0, ha)
    def __init__(self, rng, input, n_input, n_hidden, n_output):
        super(MLP_relu, self).__init__(rng, input, n_input, n_hidden,
n_output)
        self.p_y_given_x = T.nnet.softmax(T.dot(T.tanh(T.maximum(T.dot
(input, self.W1) + self.b1, 0.)), self.W2) + self.b2)
```

In [10]:
```
def run_test_relu(n_epochs,
                  n_hidden,
                  learning_rate=0.01,
                  L1_reg=0.00,
                  L2_reg=0.0001,
                  batch_size=100,
                  check_gradients=False):
    n_input = 28*28
    n_output =10

    # only check the fist batch if debug
    if check_gradients==True:
        batch_size=1

    # load the data
    datasets = load_data()
    train_set_x, train_set_y, test_set_x, test_set_y = datasets

    # compute number of minibatches for training and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0] / ba
tch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] / batc
h_size

    print '\n... building the model'

    # allocate symbolic variables for the data
    index = T.lscalar()  # index to a minibatch
    x = T.matrix('x')    # the data is presented as rasterized images
```

```
    y = T.ivector('y')   # the labels are presented as 1D vector of in
t labels

    rng = numpy.random.RandomState(1234)

    # construct the MLP class
    classifier = MLP_relu(rng, x, n_input, n_hidden, n_output)

    # minimize negative log likelihood of the model
    # and the regularization terms (L1 and L2)
    # cost is expressed symbolically
    cost = (
            classifier.negative_log_likelihood(y)
            + L1_reg * classifier.L1
            + L2_reg * classifier.L2
            )

    # returns the cost
    ret_cost = theano.function(
    inputs=[index],
    outputs=cost,
    givens={
            x: train_set_x[index * batch_size:(index + 1) * batch_size
],
            y: train_set_y[index * batch_size:(index + 1) * batch_size
]
            }
        )
    # fit on train set
    check_fit_train_set = theano.function(
        inputs=[],
        outputs=classifier.errors(y),
        givens={
                x: train_set_x,
                y: train_set_y
                }
            )

    # fit on test set
    check_fit_test_set = theano.function(
        inputs=[],
        outputs=classifier.errors(y),
        givens={
                x: test_set_x,
                y: test_set_y
                }
            )

    # compile a Theano function that computes the mistakes that are ma
de
    # by the model on a minibatch of the train set (we'll see overfitt
ing)
    check_fit_batch = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
```

```
            x: train_set_x[index * batch_size:(index + 1) * batch_size
],
            y: train_set_y[index * batch_size:(index + 1) * batch_size
]
            }
        )

    # compute the gradient of cost with respect to theta (sorted in pa
rams)
    # the resulting gradients will be stored in a list gparams
    gradient_w1 = T.grad(cost, classifier.W1)
    gradient_b1 = T.grad(cost, classifier.b1)

    gradient_w2 = T.grad(cost, classifier.W2)
    gradient_b2 = T.grad(cost, classifier.b2)

    # specify how to update the parameters of the model as a list of
    # (variable, update expression) pairs
    updates = [
        (param, param - learning_rate * gradient)
        for param, gradient in [(classifier.W1, gradient_w1),
                                (classifier.b1, gradient_b1),
                                (classifier.W2, gradient_w2),
                                (classifier.b2, gradient_b2)]]

    values = [gradient_w1, gradient_b1, gradient_w2, gradient_b2]

    ret_gradient = theano.function(
        inputs=[index],
        outputs=values,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_siz
e],
            y: train_set_y[index * batch_size: (index + 1) * batch_siz
e]
        }
    )

    # updates the parameter of the model based on the rules defined in
 'updates'
    train_model = theano.function(
        inputs=[index],
        outputs=[],
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_siz
e],
            y: train_set_y[index * batch_size: (index + 1) * batch_siz
e]
        }
    )

    print '... training'

    best_iter = 0
    test_scores = []
```

```python
    train_scores = []

    epoch = 0


    while (epoch < n_epochs):

        epoch = epoch + 1

        for minibatch_index in xrange(n_train_batches):
            train_model(minibatch_index)
            if check_gradients==True:
                symbolic_gradients = ret_gradient(minibatch_index)

                epsilon = 1E-4
                a = numpy.zeros((n_input, n_hidden))
                b = numpy.zeros((n_hidden,))
                c = numpy.zeros((n_hidden, n_output))
                d = numpy.zeros((n_output,))
                grad=  [a,b,c,d]

                # b1
                b1_vals = classifier.b1.get_value()
                for i in range(len(grad[1])):
                    # compute low
                    b1_vals[i] -= epsilon
                    classifier.b1.set_value(b1_vals, borrow=True)
                    low= ret_cost(minibatch_index)

                    # compute high
                    b1_vals[i] += 2.*epsilon
                    classifier.b1.set_value(b1_vals, borrow=True)
                    high= ret_cost(minibatch_index)

                    # reset the value
                    b1_vals[i] -= epsilon
                    # store the gradient
                    grad[1][i] = (high - low) / (2.*epsilon)

                # b2
                b2_vals = classifier.b2.get_value()
                for i in range(len(grad[3])):
                    # compute low
                    b2_vals[i] -= epsilon
                    classifier.b2.set_value(b2_vals, borrow=True)
                    low= ret_cost(minibatch_index)

                    # compute high
                    b2_vals[i] += 2.*epsilon
                    classifier.b2.set_value(b2_vals, borrow=True)
                    high= ret_cost(minibatch_index)

                    # reset the value
                    b2_vals[i] -= epsilon

                    # store the gradient
```

```
                    grad[3][i] = (high - low) / (2.*epsilon)


                # W1
                w1_vals = classifier.W1.get_value()
                for i in range(len(grad[0])):
                    for j in range(len(grad[0][0])):

                        # compute low
                        w1_vals[i][j] -= epsilon
                        classifier.W1.set_value(w1_vals, borrow=True)
                        low= ret_cost(minibatch_index)

                        # compute high
                        w1_vals[i][j] += 2.*epsilon
                        classifier.W1.set_value(w1_vals, borrow=True)
                        high= ret_cost(minibatch_index)

                        # reset the value
                        w1_vals[i][j] -= epsilon

                        # store the gradient
                        grad[0][i][j] = (high - low) / (2.*epsilon)

                # W2
                w2_vals = classifier.W2.get_value()
                for i in range(len(grad[2])):
                    for j in range(len(grad[2][0])):

                        # compute low
                        w2_vals[i][j] -= epsilon
                        classifier.W2.set_value(w2_vals, borrow=True)
                        low= ret_cost(minibatch_index)

                        # compute high
                        w2_vals[i][j] += 2.*epsilon
                        classifier.W2.set_value(w2_vals, borrow=True)
                        high= ret_cost(minibatch_index)

                        # reset the value
                        w2_vals[i][j] -= epsilon

                        # store the gradient
                        grad[2][i][j] = (high - low) / (2.*epsilon)

                return (symbolic_gradients, grad)

        # verify the fit on the datasets
        train_scores.append(check_fit_train_set())
        test_scores.append(check_fit_test_set())


    print '... done\n'
    return (train_scores, test_scores)
```

In [11]:
```
train_scores, test_scores = run_test_relu(1000, 500)
```

```
... building the model
... training
... done
```

In [12]:
```
plt.plot(train_scores, label='train set error')
plt.plot(test_scores, label='test set error')
plt.xlabel('epoch')
plt.ylabel('error rate')
plt.grid(True)
plt.legend()
plt.title('Error rates for RELU (500 hidden,1000 epochs)')
plt.show()
```