M3C 2018 Homework 2 Solution

In this assignment you will implement and analyze two "neural" classification methods which will be used to classify images of handwritten digits as even (y=0) or odd (y=1). The two methods to be investigated are: 1) the single neuron model (SNM) and 2) a neural network model (NNM) with one internal layer and a single neuron in its output layer. Notes describing these models have been posted online and should be read carefully before proceeding with this assignment.

Part 1: Single Neuron Model

1) (20 pts) Complete the subroutine *snmodel* in *hw2_dev.f90* so that it computes the cost function, c, and its gradient, ∇c, for the SNM with fitting parameters (the weight vector and bias) provided as input to the subroutine. The least-squares cost described in the NN notes should be used. Note that the input data and labels should be set elsewhere before the subroutine is called.

Ans: The code is shown below. Recognizing matrix multiplication and use *matmul* when computing z and dc/dw, simplifies the code.

```
!Compute cost function and its gradient for single neuron model
!for d images (in nm_x) and d labels (in nm_y) along with the
!fitting parameters provided as input in fvec.
!The weight vector, w, corresponds to fvec(1:n) and
!the bias, b, is stored in fvec(n+1)
!Similarly, the elements of dc/dw should be stored in cgrad(1:n)
!and dc/db should be stored in cgrad(n+1)
!Note: nm_x and nm_y must be allocated and set before calling this subroutine.
subroutine snmodel(fvec,n,d,c,cgrad)
  implicit none
  integer, intent(in) :: n,d !training data sizes
  real(kind=8), dimension(n+1), intent(in) :: fvec !fitting parameters
  real(kind=8), intent(out) :: c !cost
  real(kind=8), dimension(n+1), intent(out) :: cgrad !gradient of cost
  !Declare other variables as needed
  real(kind=8) :: dinv,dcdb
  real(kind=8),dimension(d) :: z,a,e,gamma,dadb,eg
  real(kind=8), dimension(n) :: dcdw

  dinv = 1.d0/dble(d)

  !Compute inner layer activation
  z = matmul(fvec(1:n),nm_x) + fvec(n+1)
  a = 1.d0/(1.d0+exp(-z))

  !Compute cost
  gamma = a*(1.d0-a)
  e = a-nm_y
  c = 0.5d0*dinv*sum(e**2)

  !Compute gradient of cost
  eg = e*gamma
  cgrad(n+1) = dinv*sum(eg) !dcdb
  cgrad(1:n) = dinv*matmul(nm_x,eg) !dcdw

end subroutine snmodel
```

2) (15 pts) Complete the function *snm_test* in *hw2_dev.py* so that it (a) trains the SNM by finding the fitting parameters that minimize the cost function computed in *snmodel*, and (b) computes the testing error after the model has been trained and the optimal fitting parameters have been found. The fitting parameters obtained after training and the testing error should both be returned by the function. The input parameter, *omethod* should set the optimizer that is used. When *omethod=1*, *l-bfgs-b* should be used to minimize the cost, and when *omethod=2*, the SGD routine included in the *nmodel* module should be used. In both cases, the cost and gradient should be evaluated using *snmodel*. The initial fitting parameters should be sampled from a random normal distribution. The testing data throughout this assignment should consist of the final 1e4 images contained in the provided dataset.

Ans: The required code is below. Using *jac=True* with minimize allows the *l-bfgs-b* solver to use the gradient of the cost function.

```
def snm_test(X,y,X_test,y_test,omethod,input=(None)):
    """Train single neuron model with input images and labels (i.e. use data in X and y), then comput
    using X_test, y_test. The fitting parameters obtained via training should be returned in the 1-d
    X: training image data, should be 784 x d with 1<=d<=60000
    y: training image labels, should contain d elements
    X_test,y_test: should be set as in read_data above
    omethod=1: use l-bfgs-b optimizer
    omethod=2: use stochastic gradient descent
    input: tuple, set if and as needed
    """
    n = X.shape[0]
    fvec = np.random.randn(n+1) #initial fitting parameters

    #Add code to train SNM and evaluate testing test_error
    d = X.shape[1]
    nm.nm_x = X
    nm.nm_y = y

    #Train snm using appropriate model
    if omethod==1:
        res = minimize(nm.snmodel,fvec,args=(d,),method='L-BFGS-B',jac=True)
        fvec_f = res.x
    elif omethod==2:
        fvec_f = nm.sgd(fvec,n,0,d,0.1)
    else:
        print("error, omethod must be 1 or 2")
        return None

    #Compute testing error
    z = np.dot(fvec_f[:-1],X_test) + fvec_f[-1]
    a_int = np.round(1.0/(1.0+np.exp(-z)))
    eps = np.abs(a_int - y_test)
    test_error = eps.sum()/y_test.size

    output = (None) #output tuple, modify as needed
    return fvec_f,test_error,output
#--------------------------------------------
```

3) (5 pts) In the docstring for *snm_test*, discuss the expected advantages and/or disadvantages of training and testing the single neuron model purely in Python vs. the Python+Fortran approach used here. Pick 2 key points and provide clear concise explanations.

Ans: The two key points to consider are 1) ease of code development and 2) speed/efficiency. It is clearly easier to develop code in Python (due to the presence of a terminal and the lack of a compilation step), so the question then is, is the Fortran+Python approach faster? The testing error and *snmodel* calculations can be thoroughly vectorized, so the only gain in speed will be with the use of SGD in Fortran which cannot be written in vectorized form in Python and will thus be slower. Note that *l-bfgs-b* in Scipy is Fortran code compiled with *f2py*.

Part 2: Neural Network Model

1) (30 pts) Complete the subroutine *nnmodel* in *hw2_dev.f90* so that it computes the cost function, *c*, and its gradient, ∇c. for the NNM with fitting parameters (the weight matrix and bias vector) provided as input to the subroutine. The least-squares cost described in the NN notes should again be used. The number of neurons in the inner layer is set by the input parameter, *m*, and as noted above, there should be one output neuron. As with *snmodel*, the input data and labels should be set elsewhere before the subroutine is called.

Ans: The required code below.

```
!!Compute cost function and its gradient for neural network model
!for d images (in nm_x) and d labels (in nm_y) along with the
!fitting parameters provided as input in fvec. The network consists of
!an inner layer with m neurons and an output layer with a single neuron.
!fvec contains the elements of dw_inner, b_inner, w_outer, and b_outer
! Code has been provided below to "unpack" fvec
!The elements of dc/dw_inner,dc/db_inner, dc/dw_outer,dc/db_outer should be stored in cgrad
```

```
    !and should be "packed" in the same order that fvec was unpacked.
    !Note: nm_x and nm_y must be allocated and set before calling this subroutine.
    subroutine nnmodel(fvec,n,m,d,c,cgrad)
      implicit none
      integer, intent(in) :: n,m,d !training data and inner layer sizes
      real(kind=8), dimension(m*(n+2)+1), intent(in) :: fvec !fitting parameters
      real(kind=8), intent(out) :: c !cost
      real(kind=8), dimension(m*(n+2)+1), intent(out) :: cgrad !gradient of cost
      integer :: i1,j1,l1
      real(kind=8), dimension(m,n) :: w_inner
      real(kind=8), dimension(m) :: b_inner,w_outer
      real(kind=8) :: dinv,b_outer
      !Declare other variables as needed
      real(kind=8), dimension(m,d) :: z_inner,a_inner,g_inner
      real(kind=8), dimension(d) :: z_outer,a_outer,e_outer,g_outer,eg_outer
      real(kind=8) :: dcdb_outer
      real(kind=8),dimension(m) :: dcdw_outer
      real(kind=8), dimension(m) :: dcdb_inner
      real(kind=8), dimension(m,n) :: dcdw_inner

      dinv = 1.d0/dble(d)

      !unpack fitting parameters (use if needed)
      do i1=1,n
        j1 = (i1-1)*m+1
        w_inner(:,i1) = fvec(j1:j1+m-1) !inner layer weight matrix
      end do
      b_inner = fvec(n*m+1:n*m+m) !inner layer bias vector
      w_outer = fvec(n*m+m+1:n*m+2*m) !output layer weight vector
      b_outer  = fvec(n*m+2*m+1) !output layer bias

      !Add code to compute c and cgrad

      !Compute inner layer activation vector, a_inner
      z_inner = matmul(w_inner,nm_x)
      do i1=1,d
        z_inner(:,i1) = z_inner(:,i1) + b_inner
      end do
      a_inner = 1.d0/(1.d0 + exp(-z_inner))

      !Compute outer layer activation (a_outer) and cost
      z_outer = matmul(w_outer,a_inner) + b_outer
      a_outer = 1.d0/(1.d0+exp(-z_outer))
      e_outer = a_outer-nm_y
      c = 0.5d0*dinv*sum((e_outer)**2)

      !Compute dc/db_outer and dc/dw_outer
      g_outer = a_outer*(1.d0-a_outer)
      eg_outer = e_outer*g_outer
      dcdb_outer = dinv*sum(eg_outer)
      dcdw_outer = dinv*matmul(a_inner,eg_outer)

      !Compute dc/db_inner and dc/dw_inner
      g_inner = a_inner*(1.d0-a_inner)
      dcdb_inner = dinv*w_outer*matmul(g_inner,eg_outer)
      do l1 = 1,n
        dcdw_inner(:,l1) = dinv*w_outer*matmul(g_inner,(nm_x(l1,:)*eg_outer))
      end do

      !Pack gradient into cgrad
      do i1=1,n
        j1 = (i1-1)*m+1
        cgrad(j1:j1+m-1) = dcdw_inner(:,i1)
      end do
      cgrad(n*m+1:n*m+m) = dcdb_inner
      cgrad(n*m+m+1:n*m+2*m) = dcdw_outer
      cgrad(n*m+2*m+1) = dcdb_outer

    end subroutine nnmodel
```

2) (5 pts) Complete the function *nnm_test* in *hw2_dev.py* so that it (a) trains the NNM by finding the fitting parameters that minimize the cost function computed in *nnmodel*, and (b) computes the testing error after the model has been trained and the optimal fitting parameters have been found. The fitting parameters obtained after training and the testing error should both be returned by the function. The input parameter, *omethod* should set the optimizer that is used. When *omethod=1*, *l-bfgs-b* should be used to minimize the cost, and when *omethod=2*, the SGD routine included in the *nmodel* module should be used. In both cases, the cost and gradient should be evaluated using *nnmodel*. The initial fitting parameters should be sampled from a random normal distribution. The testing data again should consist of the final 1e4 images contained in the provided dataset and the corresponding labels.

Ans: The training code is below.

```
def nnm_test(X,y,X_test,y_test,m,omethod,input=(None)):
    """Train neural network model with input images and labels (i.e. use data in X and y), then compu
    using X_test, y_test. The fitting parameters obtained via training should be returned in the 1-d
    X: training image data, should be 784 x d with 1<=d<=60000
    y: training image labels, should contain d elements
    X_test,y_test: should be set as in read_data above
    m: number of neurons in inner layer
    omethod=1: use l-bfgs-b optimizer
    omethod=2: use stochastic gradient descent
    input: tuple, set if and as needed
    """
    n = X.shape[0]
    fvec = np.random.randn(m*(n+2)+1) #initial fitting parameters

    #Add code to train NNM and evaluate testing error, test_error
    d = X.shape[1]
    nm.nm_x = X.copy()
    nm.nm_y = y.copy()

    #Train snm using appropriate model
    if omethod==1:
        res = minimize(nm.nnmodel,fvec,args=(n,m,d),method='L-BFGS-B',jac=True)
        fvec_f = res.x
    elif omethod==2:
        fvec_f = nm.sgd(fvec,n,m,d,0.1)
    else:
        print("error, omethod must be 1 or 2")
        return None

    nm.nm_x = X_test.copy()
    nm.nm_y = y_test.copy()

    test_error = nm.run_nnmodel(fvec_f,n,m,d)

    output = (None) #output tuple, modify as needed
    return fvec_f,test_error,output
#--------------------------------------------
```

I have put the testing error calculation in a Fortran subroutine which should be a little more efficient than a Python implementation.

```
!Compute test error provided fitting parameters
!and with testing data stored in nm_x_test and nm_y_test
subroutine run_nnmodel(fvec,n,m,d,test_error)
  implicit none
  integer, intent(in) :: n,m,d !training data and inner layer sizes
  real(kind=8), dimension(m*(n+2)+1), intent(in) :: fvec !fitting parameters
  real(kind=8), intent(out) :: test_error !test_error
  integer :: i1,j1
  real(kind=8), dimension(m,n) :: w_inner
  real(kind=8), dimension(m) :: b_inner,w_outer
  real(kind=8) :: b_outer
  !Declare other variables as needed
  real(kind=8), dimension(m,d) :: z_inner,a_inner
  real(kind=8), dimension(d) :: z_outer,a_outer
  integer, dimension(d) :: e_outer
```

```
    !unpack fitting parameters (use if needed)
    do i1=1,n
      j1 = (i1-1)*m+1
      w_inner(:,i1) = fvec(j1:j1+m-1) !inner layer weight matrix
    end do
    b_inner = fvec(n*m+1:n*m+m) !inner layer bias vector
    w_outer = fvec(n*m+m+1:n*m+2*m) !output layer weight vector
    b_outer  = fvec(n*m+2*m+1) !output layer bias


    !Compute inner layer activation vector, a_inner
    z_inner = matmul(w_inner,nm_x)
    do i1=1,d
      z_inner(:,i1) = z_inner(:,i1) + b_inner
    end do
    a_inner = 1.d0/(1.d0 + exp(-z_inner))

    !Compute outer layer activation (a_outer) and cost
    z_outer = matmul(w_outer,a_inner) + b_outer
    a_outer = 1.d0/(1.d0+exp(-z_outer))
    e_outer = nint(a_outer-nm_y)

    test_error = dble(sum(e_outer))/dble(d)

  end subroutine run_nnmodel
```

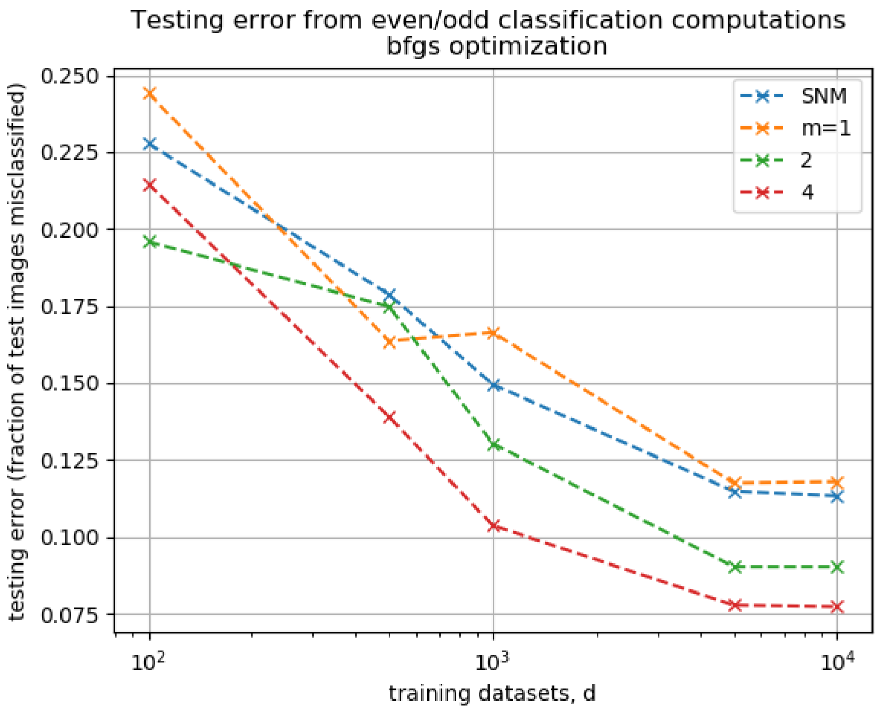
3) (25 pts) Analyze and compare the performance of the single neuron and neural network model training codes as well as the performance of the models themselves when applied to the even/odd classification problem. You should keep the test data fixed to 10000 images and it is sufficient to keep the SGD learning rate set to $\alpha = 0.1$. Otherwise, it is up to you how to vary parameters and investigate this problem. Add python code to the function *nm_analyze* in *hw2_dev.py* which produces 2-4 figures which illustrate the most important qualitative trends you identify. The docstring of *nm_analyze* should contain a clear, concise discussion of these trends and your main conclusions. Save your figures as *.png* files with the names *hw21.png*, *hw22.png*, …, and submit them with your codes.

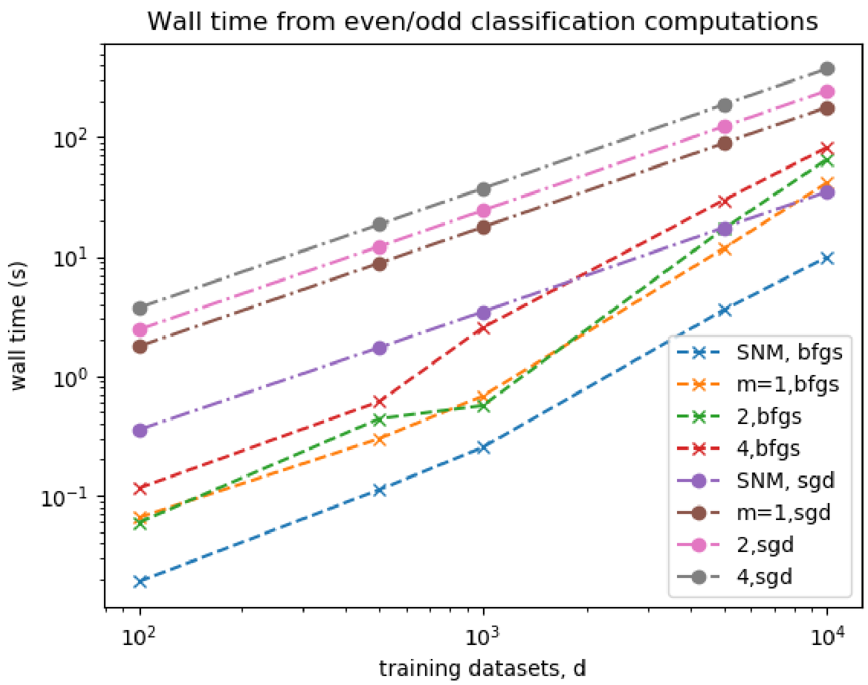
Ans: The Python function, *nm_analyze*, is below. The analysis focuses on the training time and the testing error as the number of training images, the number of inner-layer neurons, and the choice of optimizer are all varied. Note that rather than using the *snm_test* and *nnm_test* functions, I have chosen to copy the relevant portions of those functions into *nm_analyze* in order to ease the load of calculations with large matrices. We expect that computation time will increase for larger values of m and d, and choose these parameters accordingly to ensure that the time is not excessive and that key trends may be observed.

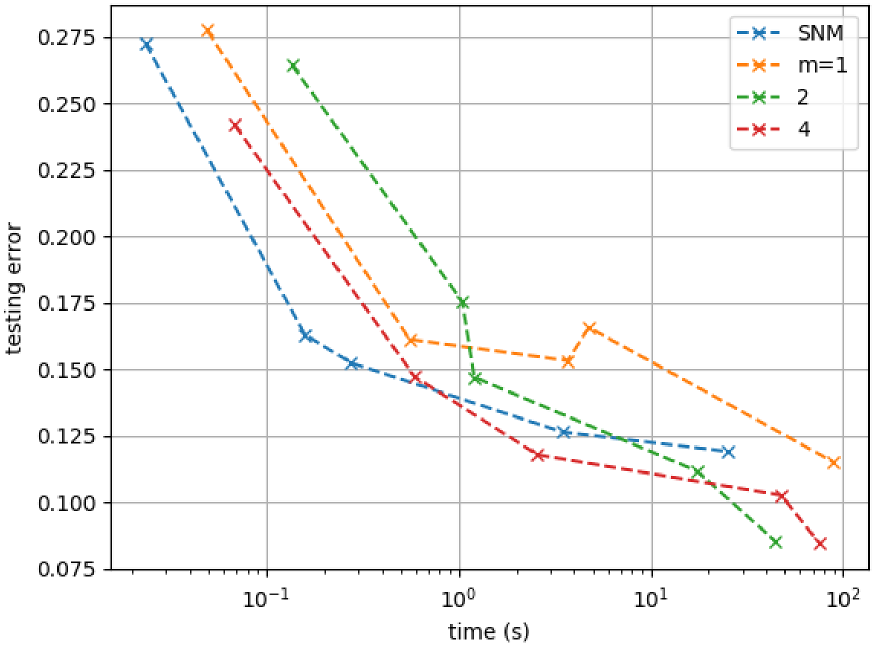
The first figure below shows the testing error with *l-bfgs-b* as m and d is varied. We see that there is a initially rapid drop in the error before plateauing as d approaches *1e4*. The level of this plateau depends on m – as m is increased the testing error tends to decrease though this trend is less clear at smaller values of d. *SNM* and *NNM* with *m=1* show similar errors, but as we see below, *SNM* is considerably faster.

The second figure shows the time required by both optimizers for the same range of m and d as for the first figure. There is a linear increase in time as d increases for the *SGD* calculations, however *l-bfgs-b* shows more complicated behavior, particularly when more images are used. This can be expected – with larger d, the optimization problem becomes more challenging, though this is not recognized by *SGD* which runs a fixed number of iterations and doesn't test for convergence.

The final figure compares the error and time for the l-bfgs-b calculations (it is straightforward to make the equivalent for SGD). The "preferred" model depends on the desired accuracy. If a testing error of about 15% or greater is acceptable, then the SNM should clearly be used. For higher accuracies, the figure broadly indicates that larger values of m provide superior results, however, the *m=2* results do seem to "overtake" the *m=4* results at larger d (and lower error) – further investigation (and averaging over multiple trials!) is needed to provide a precise explanation of this behavior, however it is likely due to greater sensitivity of l-bfgs-b to the quality of the guess when larger numbers of fitting parameters need to be set. There are other interesting avenues which could also have been pursued, for example, how do the testing errors and wall times vary when larger values of m (m>4) are used?


Testing error from even/odd classification computations
bfgs optimization
testing error (fraction of test images misclassified)
training datasets, d
0.250
0.225
0.200
0.175
0.150
0.125
0.100
0.075
10^2
10^3
10^4
SNM
m=1
2
4


Wall time from even/odd classification computations
wall time (s)
training datasets, d
10^2
10^1
10^0
10^{-1}
10^2
10^3
10^4
SNM, bfgs
m=1,bfgs
2,bfgs
4,bfgs
SNM, sgd
m=1,sgd
2,sgd
4,sgd




```
def nm_analyze(mvalues,dvalues):
    """ Analyze performance of single neuron and neural network models
    on even/odd image classification problem
    Add input variables and modify return statement as needed.
    Should be called from
    name==main section below
    """

    #Read in data, and initialize arrays
    Xfull,yfull,X_test,y_test = read_data()
    dt = np.zeros((len(dvalues),len(mvalues),2))
    e_test = np.zeros((len(dvalues),len(mvalues),2))
    n = Xfull.shape[0]

    nm.nm_x_test = X_test
    nm.nm_y_test = y_test
    d_test = y_test.size

    #Loop through d and m values storing wall time and testing error
    for i,d in enumerate(dvalues):
        nm.nm_x = Xfull[:,:d]
        nm.nm_y = yfull[:d]
        for j,m in enumerate(mvalues):
            for k,omethod in enumerate([1,2]):
                print("i,j,k=",i,j,k)
                args_net = (n,m,d)
                if m==0: #SNM
                    fvec = np.random.randn(n+1)
                    args = (d,)
                    if omethod==1: #BFGS
                        t1=time()
                        res = minimize(nm.snmodel,fvec,args,method='L-BFGS-B',jac=True)
                        fvec_f = res.x
                        t2=time()

                    else: #SGD
                        t1=time()
                        fvec_f = nm.sgd(fvec,n,m,d,0.1)
                        t2 = time()
                    #Compute testing error
                    z = np.dot(fvec_f[:-1],X_test) + fvec_f[-1]
                    a_int = np.round(1.0/(1.0+np.exp(-z)))
                    eps = np.abs(a_int - y_test)
                    e_test[i,j,k] = eps.sum()/y_test.size
```

```
                    else: #NNM
                        fvec = np.random.randn(m*(n+2)+1)
                        args = (n,m,d)
                        if omethod==1: #BFGS
                            t1=time()
                            res = minimize(nm.nnmodel,fvec,args,method='L-BFGS-B',jac=True)
                            fvec_f = res.x
                            t2=time()

                        else: #SGD
                            t1=time()
                            fvec_f = nm.sgd(fvec,n,m,d,0.1)
                            t2 = time()

                        e_test[i,j,k] = nm.run_nnmodel(fvec_f,n,m,d_test)
                    dt[i,j,k] = t2-t1


        plt.figure()
        plt.semilogx(dt,e_test[:,:,0],'x--')
        plt.xlabel('time (s)')
        plt.ylabel('testing error')
        plt.title('Testing error vs. wall time for l-bfgs-b classification calculations \n d=%s' %str(dva
        plt.legend(('SNM','m=1','2','4'))

        return (e_test,dt)
    #---------------------------------------------
```