The following block is for the network builder python class

Note that i am also attaching the file network.py

In [1]:
```
%%writefile network.py

import numpy as np
import scg as scg
from copy import copy

class NetworkModeler:

    def __init__(self, nInputAttributes, hiddenLayersSpec, numOutputs):
        try:
            inputAndHiddenLayersSpecList = [nInputAttributes] + list(hiddenLayersSpec)
            hiddenLayersSpecList = list(hiddenLayersSpec)
        except:
            inputAndHiddenLayersSpecList = [nInputAttributes] + [hiddenLayersSpec]
            hiddenLayersSpecList = [hiddenLayersSpec]

        self.Vs = [] # list of matrices of hidden layer weights

        for i in range(len(inputAndHiddenLayersSpecList)-1):
            sqrtOfCols = np.sqrt(inputAndHiddenLayersSpecList[i])
            V=1/sqrtOfCols * np.random.uniform(-1, 1, size=(1+inputAndHiddenLayersSpecList[i], inputAndHiddenLayersSpe
            self.Vs.append(V)

        # lone output layer weight matrix
        self.W = 1/np.sqrt(hiddenLayersSpecList[-1]) * np.random.uniform(-1, 1, size=(1+hiddenLayersSpecList[-1], num(

        self.nInputAttributes, self.hiddenLayersSpecList, self.numOutputs = nInputAttributes, hiddenLayersSpecList, nu

        self.Xmeans = None
        self.Xstds = None
        self.Tmeans = None
        self.Tstds = None
        self.trained = False
        self.reason = None
        self.errorTrace = None
        self.numberOfIterations = None

    def __repr__(self):
        str = 'Network({}, {}, {})'.format(self.nInputAttributes, self.hiddenLayersSpecList, self.numOutputs)
        # str += '  Standardization parameters' + (' not' if self.Xmeans == None else '') + ' calculated.'
        if self.trained:
            str += '\n   Network was trained for {} iterations. Final error is {}.'.format(self.numberOfIterations,
                                                                                           self.errorTrace[-1])
        else:
            str += '  Network is not trained.'
        return str

    def standardizeX(self, X):
        result = (X - self.Xmeans) / self.XstdsFixed
        result[:, self.Xconstant] = 0.0
        return result

    def unstandardizeX(self, Xs):
        return self.Xstds * Xs + self.Xmeans

    def standardizeT(self, T):
        result = (T - self.Tmeans) / self.TstdsFixed
        result[:, self.Tconstant] = 0.0
        return result

    def unstandardizeT(self, Ts):
        return self.Tstds * Ts + self.Tmeans

    def getSCGWtVectorFromWtMatrices(self, Vs, W):
        return np.hstack([V.flat for V in Vs] + [W.flat])

    def getWtMatricesFromSCGWtVector(self,w):
        first = 0
        nhs = self.hiddenLayersSpecList
        numInThisLayer = self.nInputAttributes # start with inputs
        for i in range(len(self.Vs)):
            wtVectorItem = w[first:first+(numInThisLayer+1)]
            self.Vs[i][:] =w[first:first+(numInThisLayer+1) * self.hiddenLayersSpecList[i]].reshape((numInThisLayer+1,
                                                                                                     self.hiddenLayersSpecl

            first += (numInThisLayer+1) * self.hiddenLayersSpecList[i]
            numInThisLayer = self.hiddenLayersSpecList[i]
```

```python
        self.W[:] = w[first:].reshape((numInThisLayer+1, self.numOutputs))

    # takes in the input matrix, Output matrix
    # runs nIterations of scaled conjugate gradient descent
    # arrives at the best matrices of hidden layer weights and output layer weight matrix
    # at the end the best weights for each hidden layer and output layer are available in list of hidden layer weigh
    #likewise the best weights for output layer are available in output layer weight matrix
    def trainBySCG(self, X, T, nIterations=100, verbose=False, weightPrecision=0, errorPrecision=0, saveWeightsHistory

        if self.Xmeans is None:
            self.Xmeans = X.mean(axis=0)
            self.Xstds = X.std(axis=0)
            self.Xconstant = self.Xstds == 0
            self.XstdsFixed = copy(self.Xstds)
            self.XstdsFixed[self.Xconstant] = 1
        X = self.standardizeX(X)

        if T.ndim == 1:
            T = T.reshape((-1,1))

        if self.Tmeans is None:
            self.Tmeans = T.mean(axis=0)
            self.Tstds = T.std(axis=0)
            self.Tconstant = self.Tstds == 0
            self.TstdsFixed = copy(self.Tstds)
            self.TstdsFixed[self.Tconstant] = 1
        T = self.standardizeT(T)

        ## takes in flattened weight vector with minimized error function from previous backward pass
        ## returns the mse error function using neural network forward pass
        def errorFunctionOfWts(w):
            self.getWtMatricesFromSCGWtVector(w)
            Zprev = X
            for i in range(len(self.hiddenLayersSpecList)):
                V = self.Vs[i]
                # invoke hyperbolic tangent function in each hidden layer
                Zprev = np.tanh(Zprev @ V[1:,:] + V[0:1,:])  # handling bias weight without adding column of 1's
            Y = Zprev @ self.W[1:,:] + self.W[0:1,:]
            return np.mean((T-Y)**2)

        ## takes in flattened weight vector with minimized error function from previous backward pass
        ## runs descent and returns new flattened weight vector with minimized error function from this backward pass
        def gradientOfErrorFunctionOfWts(w):
            ## get new weights from last run of SCG
            self.getWtMatricesFromSCGWtVector(w)
            Zprev = X
            Z = [Zprev]
            for i in range(len(self.hiddenLayersSpecList)):
                V = self.Vs[i]
                Zprev = np.tanh(Zprev @ V[1:,:] + V[0:1,:])
                Z.append(Zprev)
            Y = Zprev @ self.W[1:,:] + self.W[0:1,:]
            delta = -(T - Y) / (X.shape[0] * T.shape[1])
            dW = 2 * np.vstack(( np.ones((1,delta.shape[0])) @ delta,
                                 Z[-1].T @ delta ))
            dVs = []
            delta = (1 - Z[-1]**2) * (delta @ self.W[1:,:].T)
            for Zi in range(len(self.hiddenLayersSpecList), 0, -1):
                Vi = Zi - 1 # because X is first element of Z
                dV = 2 * np.vstack(( np.ones((1,delta.shape[0])) @ delta,
                                     Z[Zi-1].T @ delta ))
                dVs.insert(0,dV)
                delta = (delta @ self.Vs[Vi][1:,:].T) * (1 - Z[Zi-1]**2)
            # return the latest minimized error function weights packed as a flat vector
            return self.getSCGWtVectorFromWtMatrices(dVs, dW)

        scgresult = scg.scg(self.getSCGWtVectorFromWtMatrices(self.Vs, self.W),
                            errorFunctionOfWts, gradientOfErrorFunctionOfWts,
                            xPrecision = weightPrecision,
                            fPrecision = errorPrecision,
                            nIterations = nIterations,
                            verbose=verbose,
                            ftracep=True,
                            xtracep=saveWeightsHistory)

        self.getWtMatricesFromSCGWtVector(scgresult['x'])
        self.reason = scgresult['reason']
```

```
        self.errorTrace = np.sqrt(scgresult['ftrace']) # * self.Tstds # to unstandardize the MSEs
        self.numberOfIterations = len(self.errorTrace)
        self.trained = True
        self.weightsHistory = scgresult['xtrace'] if saveWeightsHistory else None
        return self

    def predict(self, X, allOutputs=False):
        Zprev = self.standardizeX(X)
        Z = [Zprev]
        for i in range(len(self.hiddenLayersSpecList)):
            V = self.Vs[i]
            Zprev = np.tanh( Zprev @ V[1:,:] + V[0:1,:])
            Z.append(Zprev)
        Y = Zprev @ self.W[1:,:] + self.W[0:1,:]
        Y = self.unstandardizeT(Y)
        return (Y, Z[1:]) if allOutputs else Y

    def getNumberOfIterations(self):
        return self.numberOfIterations

    def getErrors(self):
        return self.errorTrace

    def getWeightsHistory(self):
        return self.weightsHistory
```

```
Overwriting network.py
```

Note how we import the crucial numpy dependency and our network class

In [2]:
```
import numpy as np
import network as nn
import imp
imp.reload(nn)
```

Out[2]:
```
<module 'network' from 'C:\\Users\\santanu\\pycoursework\\network.py'>
```

Method to train on training dataset.

We will be repeatedly passing this function down to our cycles of train, validate, test

Training set is :

- matrix of inputs X minus the output/target columns
- and output/target columns expressed as target matrix T

Method takes arguments X, T and a 2 element neural network parameters list 'parameters'.

- 1st element of 'parameters' is another positional list of hidden layer specs

  i.e # of units in each hidden layer
- 2nd element of 'parameters' is number of iterations to perform in

  gradient descent using scaled conjugate gradient descent

Returns the trained neural network as model.

Returned network or model be used to predict on test set or validation set

PLEASE COPY OVER THIS METHOD in some .py file

In [3]:
```
## Method to train on training dataset.
## Training set is matrix of inputs X minus the output/target columns and output/target columns expressed as target mo
## or target column and output matrix
## Takes arguments X, T and a 2 element neural network parameters list 'parameters'
## 1st element of 'parameters' is another positional list of hidden layer specs i.e # of units in each hidden layer
## 2nd element of 'parameters' is number of iterations to perform in gradient descent using scaled conjugate gradient
## Returns the trained neural network as model that can be used to predict on test set or validation set

def trainNetwork(X,T,parameters): #X,T,[[10,10], 200]

    ## NeworkModeler object init with numInputAttributes, list of hidden layer specs
    nnet = nn.NetworkModeler(X.shape[1], parameters[0], 1 )

    ## trainBySCG on the network modeler object passing the number of iterations todo in SCG
    nnet.trainBySCG(X, T, nIterations=parameters[1], verbose=False)
    return {'neuralnetwork':nnet}
```

Method to evaluate the error (RMSE) of predictions on test set or validation set or any other dataset.

We will be repeatedly passing this function down to our cycles of train, validate, test.

This Method Takes as arguments the test set X and T matrices along with the model object.

Model object is really the constructed neural network.

- Model is used to use to predict Y matrix of outputs for input matrix X
- Predicted Y matrix of outputs is compared with recorded T matrix of outputs for the input dataset

Method eventually calculates RMSE is for Y-T diff and returns the same.

PLEASE COPY OVER THIS METHOD in some .py file

In [4]:
```
# Method to evaluate the error (RMSE) of predictions on test set or validation set or any other dataset.
# Takes as arguments the test set X and T matrices along with the model object which is really the constructed neural
# Model is used to use to predict Y matrix of outputs for input matrix X
# The predicted Y matrix of outputs is compared against recorded T matrix of outputs for the input dataset
# RMSE is calculated for Y-T diff and returned
def evaluateNetwork(model,X,T):
    Y=model['neuralnetwork'].predict(X)
    return np.sqrt(np.mean((Y-T)**2))
```

In our SPARK setup we would like to leverage the logic in this method to distribute the computes. This method is an example of what we sould do to divide the input data set into k*(k-1) folds K = test folds K-1 = validation folds For say nFolds = 5, we will try 5*4 neural networks with different random weights at each hidden layer and lone output layer Note that the number of hidden layers and #units in each of those is passed in the argument 'parameters' list's 1st element Also note that the number of iterations to do SCG is passed in the 'parameters' list's 2nd element

For distribution, for example, we would like to iterate over not only folds but also various parameter choices

- the number of hidden layers and their #units
- number of iterations in scaled conjugate gradient

We could also potentially distribute the compute across test folds or in even more nirvana case over each validation folds inside a test fold. Combination of parameter choices and folds would be perhaps the best leverage points for distributing the compute

I COULDN'T CREATE ANOTHER .py FILE for this function. Copy from here and create potentially.

In [5]:

```python
## This method does following pseudo code
##for each of the K test folds:
##     for each K-1 validation folds for this test fold:
##         instantiate neural network using supplied hidden layers parameters
##         run SCG for supplied nIterations to train the network
##         evaluate error of prediction for this validation fold by predicting using the trained network
##         if this validation fold's prediction error is < minimum error across this test fold's validation folds:
##             update new best Model i.e. the best network to be this iteration's trained network
##             update new best error to be this validation fold's prediction error
##     Now use the best network across all validation folds of this test fold to predict on this test fold
##     Evaulate this test fold's prediction error and note down the error
##     note down this testfold's chosen trained network (the best across this test fold's validation folds)
##
## Return the list of dictionaries for each test fold
## dictionary for each fold has foldNumber, best network, fold's error, minimum of fold's validation fold errors
##

def trainValidateTestKFolds(trainf,evaluatef,X,T,parameters,nFolds,
                            shuffle=False,verbose=False):
    # Randomly arrange row indices
    rowIndices = np.arange(X.shape[0])
    if shuffle:
        np.random.shuffle(rowIndices)
    # Calculate number of samples in each of the nFolds folds
    nSamples = X.shape[0]
    nEach = int(nSamples / nFolds)
    if nEach == 0:
        raise ValueError("partitionKFolds: Number of samples in each fold is 0.")
    # Calculate the starting and stopping row index for each fold.
    # Store in startsStops as list of (start,stop) pairs
    starts = np.arange(0,nEach*nFolds,nEach)
    stops = starts + nEach
    stops[-1] = nSamples
    startsStops = list(zip(starts,stops))
    # Repeat with testFold taking each single fold, one at a time
    results = []
    minimumValidationError = None
    #bestModel = None
    # Iterations  over all test folds
    for testFold in range(nFolds):
        # Find best set of parameter values
        #bestParms = []
        validationFoldErrors = []

        # iterations over validation folds and train, evaluate
        for validateFold in range(nFolds):
            if testFold == validateFold:
                continue
            # trainFolds are all remaining folds, after selecting test and validate folds
            trainFolds = np.setdiff1d(range(nFolds), [testFold,validateFold])
            # Construct Xtrain and Ttrain by collecting rows for all trainFolds
            rows = []
            for tf in trainFolds:
                a,b = startsStops[tf]
                rows += rowIndices[a:b].tolist()
            Xtrain = X[rows,:]
            Ttrain = T[rows,:]

            # Construct Xvalidate and Tvalidate
            a,b = startsStops[validateFold]
            rows = rowIndices[a:b]
            Xvalidate = X[rows,:]
            Tvalidate = T[rows,:]

            # Construct Xtest and Ttest
            a,b = startsStops[testFold]
            rows = rowIndices[a:b]
            Xtest = X[rows,:]
            Ttest = T[rows,:]

            # now train and evaluate for this validation fold
            model=trainf(Xtrain,Ttrain,parameters)
            thisValidationFoldError=evaluatef(model,Xvalidate,Tvalidate)
            if minimumValidationError == None:
                minimumValidationError = thisValidationFoldError
```

```
                bestModel = model
            else:
                if thisValidationFoldError < minimumValidationError :
                    minimumValidationError = thisValidationFoldError
                    bestModel = model

        # End of iterations over validation folds and train, evaluate

        ## Now check test errors with this best model obtained across the validations folds

        a2,b2 = startsStops[testFold]
        testRows = rowIndices[a2:b2]
        NewXtest = X[testRows,:]
        NewTtest = T[testRows,:]

        testFoldError=evaluatef(bestModel,NewXtest,NewTtest)

        results.append({'testFoldNumber': testFold,'bestNetworkForTheFold':bestModel,
                        'minValidationError': minimumValidationError,'testFoldError':testFoldError })

    # End of Iterations  over all test folds
    return results
```

One example of invoking the whole cycle of train validate and test using some toy data

First let's create some data

Toy Input matrix.

In reality we will get them from input data by minusing output cols (e.g global_active_power)

In [6]:

```
## Input matrix - in reality we will get them from input dataset by subtracting
## the output columns (e.g global_active_power)
X = np.arange(100).reshape((-1, 1))
```

Toy Output matrix.

In reality we will get them from input dataset.

We will be choosing the columns that we want to consider as outputs.

(e.g. may be just 1 column matrix with global_active_power).

In [7]:

```
## Output matrix - in reality we will get them from input dataset by choosing the
## columns that we want to consider as outputs
## ( e.g. may be just 1 column matrix with global_active_power)
T = np.abs(X -50) + X
```

In [8]:

```
## Let's examine the matrxi dimensions
X.shape, T.shape
```

Out[8]:

```
((100, 1), (100, 1))
```

Now on to the cycles of train validate test using folds and parameters.

We will only (for illustrations puproses) invoke across 20 folds (5*4).

We will be using 1 choice of hidden layers and SCHG nIterations parameter.

We will do 3 hidden layers with # units of 10,2 and 10 respectively.

We will specify 100 iterations for SCG to converge.

These parameters along with folds are our leverage for distribution.

In [9]:

```
results=trainValidateTestKFolds(trainNetwork, evaluateNetwork, X, T, [[10, 2,10], 100], nFolds=5, shuffle=False)
```

In [10]:
```
results
```

Out[10]:
```
[{'bestNetworkForTheFold': {'neuralnetwork': Network(1, [10, 2, 10], 1)
     Network was trained for 101 iterations. Final error is 0.010261224567870153.},
  'minValidationError': 1.8487929302881769,
  'testFoldError': 0.24111262392393232,
  'testFoldNumber': 0},
 {'bestNetworkForTheFold': {'neuralnetwork': Network(1, [10, 2, 10], 1)
     Network was trained for 101 iterations. Final error is 0.010261224567870153.},
  'minValidationError': 1.8487929302881769,
  'testFoldError': 0.086200600511302794,
  'testFoldNumber': 1},
 {'bestNetworkForTheFold': {'neuralnetwork': Network(1, [10, 2, 10], 1)
     Network was trained for 101 iterations. Final error is 0.016262735209292015.},
  'minValidationError': 1.2155521137928358,
  'testFoldError': 5.9408743955962624,
  'testFoldNumber': 2},
 {'bestNetworkForTheFold': {'neuralnetwork': Network(1, [10, 2, 10], 1)
     Network was trained for 101 iterations. Final error is 0.026748427470796282.},
  'minValidationError': 0.84144867128193412,
  'testFoldError': 2.5526325515716994,
  'testFoldNumber': 3},
 {'bestNetworkForTheFold': {'neuralnetwork': Network(1, [10, 2, 10], 1)
     Network was trained for 101 iterations. Final error is 0.030981920492866.},
  'minValidationError': 0.63185741892865788,
  'testFoldError': 12.091447190573591,
  'testFoldNumber': 4}]
```

In [ ]: