Software package

for computing closest flip points to inputs of deep learning models.

For a given input to a model, we seek to compute the closet point to it that exactly lies on the decision boundary of the model. This package is capable to compute such points for models of binary classification and also multi-classification.

Abbreviations:

- NN: Neural Network function
- CFP: closest flip point

Trained Network saved in text files:

- Trained network should be saved in CVS format text files.
- The weight matrices should be named as W1.csv, W2.csv, ...
- The bias vectors should be named as B1.csv, B2.csv, ...
- The tuning parameters are saved in a file sigma.csv

List of functions:

Here, a brief description is provided for each function. Detailed explanations can be viewed inside the function files.

1. **readTrainedNN** - reads the trained neural network from text files and returns the network as a structure.
2. **writeTrainedNN** - writes the parameters of a trained network into text files (CSV format).
3. **pointEval** - computes the output of network for individual inputs, along with the Jacobian of output with respect to that input.
4. **CFP** - finds the closest flip point to an input.
5. **appNNtoData** - apply the NN to a dataset, identify mistakes and arrange the information in two structs for correct classifications and mistakes.
6. **flips_calc** – calculates the closest flip points for a set of datapoints.

Example procedure:

This procedure is to make you familiar with all the functions. Not all the steps are necessary to perform in a standard computation.

1. Save your trained network as text files in a directory and go to that directory.
2. Add the directory of this package to your MATLAB path.
3. Save your training set, for example, as trainD; where trainD.X contains the data points and trainD.Y contains the corresponding labels. Rows of trainD.X would correspond to data points and columns would correspond to features. Set up the testing data in testD, similar to trainD.
4. Read the network using readTrainedNN:

   net = readTrainedNN(nlayer),

   where nlayer is a scalar corresponding to the number of hidden layers in the NN.
5. Examine the network by looking at different elements of the net structure.
6. Pick a single data point and save it in x,

   e.g. x = trainD.X(1,:),

   then evaluate x with net:

   pdata = pointEval(x,net,calcJac,print_output,soft),

   with calcJac = print_output = soft = 1, meaning that you want the Jacobian matrix to be computed, the output to be printed, and the softmax to be applied.
7. Examine the output for x by looking at different elements of the pdata structure, e.g., pdata.z, pdata.zs, pdata.y, pdata.J, etc.
8. Define possible upper and lower bounds for your variables, using mybounds.up and mybounds.low. Each of them can be a simple

scalar or a vector with same size as x. If you do not have any bounds, define mybounds = [].

9. Compute the closest flip point to x using the CFP function:

   [xf,c,dist,elap_time]=CFP(x,this,that,net,x0,mybounds),

   where this is the class of x, that is the class we want to flip to, x0 is the starting point (recommended to be x).

10. If CFP is unable to find a feasible solution, it will print a negative return code at the end of its operation along with a message. In such event, the value of c will be NaN (aka. not a number).

    If CFP is successful, it will not print any specific message, and the value of c will be the value of softmax for the classes of flip, this and that.

    In any case, examine the xf (the possible solution), dist (the 2-norm distance between x and xf), elap_time (the running time of CFP), etc.

11. Possibly repeat step 9 with a different starting point, or other configurations, as you wish.

12. Algorithm, number of iterations, and optimization parameters can be modified inside the CFP. For example, you can open the CFP file and change its algorithm from “interior-point” to “active-set”.

13. Apply the net to the entire training set:

    [Lm,Lc] = appNNtoData(trainD,net),

    where Lc will contain the information about correct classifications, and Lm will contain the information about wrong classifications.

14. Compute the closest flip points for all the mistakes saved in Lm:

    Lm = flips_calc(net,trainD,processlist,startp,mybounds),

    with processlist = [], and startp = 1. The processlist is the list of specific points in Lm that we want the flip points to be computed. When processlist is passed as empty, flips_calc() will compute the flip

points for all the rows in Lm. The startp is the starting point, when its value is 1, flips_calc() will use the original inputs as the starting points, for all computations. Further details can be found inside the function files.

The new Lm obtained after this operation will contain additional information about the closest flip points, particularly in columns 5 through 7 of the Lm.raw, and in Lm.flips.

15. Examine the index of rows of Lm that CFP has failed to find a flip point:

    idx = find(isnan(Lm.raw(:,7))),

    since 7th column of Lm corresponds to values of c for the flip points.

    If idx is not empty, consider rerunning flips_calc() for idx, with a different setting.

    You can also examine the distribution of distances among the data points and look into the flip points that have a much larger distance, compared to others.

16. Examine the directions to closest flip points for the misclassifications of the training set:

    directions = Lm.flips - trainD.X(Lm.raw(:,1),:),

    possibly compute its rank, condition number, pivoted QR factorization, etc.

17. Repeat steps 13 through 16 for the testing set.