Short Distance (Human) Stepping (SDS)-Library GUI

Utku Evci

Internship Company and Department:

École polytechnique fédérale de Lausanne

Immersive Interaction Group

01.07.2015-04.09.2015

TABLE OF CONTENTS

1. INTRODUCTION 2

2. GENERAL ORGANIZATION&PACKAGES..3

3. FUNCTIONAL PARTS & HOW TO'S...4

3.1. Data Cleaning 6

3.2. Statistics Extraction & Data Visualization 3

3.3. Neural Networks Preprocessing 5

3.4. Neural Networks Training 6

3.5. Step Sequence & Trajectory Generation 8

4. SDS GUI..10

5. DISCUSSION AND FUTURE WORK...11

6. FUNCTION TABLE 13

1. INTRODUCTION

The short distance stepping (SDS) library is written during my internship at Immersive Interaction Group at EPFL. It is not a complete library and it is not well-optimized. During the process I always tried to comment on the functions and tried to build an organized structure for complex tasks. So you can read this manual to have a general idea about my program or you can also discover the code easily by investigating directly the functions. Most of the time there are enough comments to be understood.

The purpose of the library is learning human stepping behavior and different strategies among different people through feature extraction and machine learning. Neural Network structure is used for learning. I've used the 2-D trajectory data recorded by Mr. Pisupati. First I've cleaned bad recorded data manually with the help of a script and implemented a new step detection algorithm. Then I've investigated the relationship of target location-direction with total number of steps and wrote some scripts for visualization. Then I've worked on preprocessing of data, which has the most impact on the results. I've approached the problem as given situation and target what should be the next step. Then I've extracted some features to represent different strategies and finally I've implemented a GUI to visualize the output of the network. In my last weeks at EPFL, I've also tried to learn trajectory guessing. It was not as successful as step guessing because of couple of reasons which I explain at Section 5. Any pictures used in the manual also included in the manual folder.

During the manual I am going to mention step locations a lot. When human steps, we step to ground and one of our feet is relatively stationary there, while the other foot is stepping. When I mention step locations, I infer these stationary points where the foot pivots.

Abbrevetions

SDS: Short Distance Stepping

NN: Neural Network

gSeq: A special struct for guessed step sequences.

seq: Processed data saved as seq struct (stepping sequence)

cut: Split seq, such that there is a stance and next step information only.

charStruct: Includes necessary parameters for step sequence guessing.

2. GENERAL ORGANIZATION&PACKAGES

I've organized related codes into Matlab packages. Matlab package folders start with a '+' sign and the functions inside a package folder can be accessed by a dot following the package name. So generateSteps function inside +getseg folder accessed my 'getseg.generateSteps'.

The program organized into following packages:

+anim: Include functions for all kind of plotting like animating trajectories, steps
+ftfun: Feature extraction functions.
+getseq: main package for step sequence guessing. It also includes preprocessing functions for neural networks.
+itest: Includes some investigating or testing functions for various part of the program.
+myNN: Includes NN-code for learning. This code is modified version of relevant UFLDL Tutorial Code written by the team of Andrew Ng.
+nnfun: NN functions which forward propagate the input with proper parameters an normalization
+plotFun: Functions for generating different 3-D graphs with total #steps, theta and gama.
+ppro: Preprocessing funtions for clening and processing raw trajectory data
+ppro2: First set-of preprocessing functions without feature extraction (out-of-use mostly)
+pprostance: Preprocessing for stance NN.
+regtest: Package for region based data search and strategy investigation.
+rtools: Functions for transforming data representations.
+tra: Functions for trajectory NN preprocessing.

3. FUNCTIONAL PARTS & HOW TO'S

3.1. Data Cleaning

When I was investigating the data, I have realized that the data is not perfect. There are some wrong recordings where the directions are absurd or the stances are bad. I've decided to go over all data and clean these bad recordings systematically, because otherwise this erroneous and noisy data would decrease the performance of neural network.

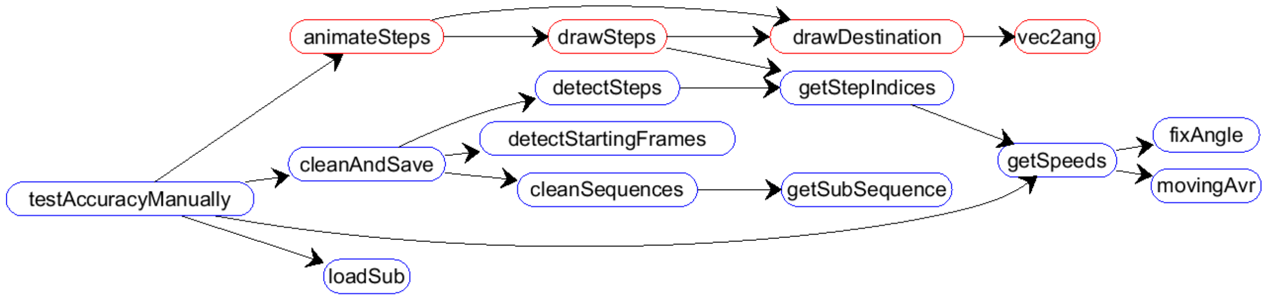
The first function called is *ppro.getAllSequences*. It loads the trajectories from *./data/* folder and saves it as array of sequences per subject *to ./dnew/* folder while normalizing 2-D location information(x,y) with the body heights of each subject.

The main script that I used for cleaning is *ppro.testAccuracyManually*, which basically ask me which subject to load selected from 1:20 and load the provided subject from the folder

./dnew/ . Then it shows the sequences with steps detected one by one and at the end of each sequence it asks me whether the sequence were right(1), wrong(0), repeat(2). If you like to quit you should type (3) and save the current progress. Then you can investigate the last sequence manually, where you may decide how much you need to cut to fix the sequence. Then you can call the script one more time and choose *alreadyLoaded* and *continue* options to continue from the last sequence. At the end of each subject(386 sequence) the script asks you to type 2xn cut matrix, where you enter the index number of sequence within 386 sequences and an integer k which indicates the first k frames to be cut. Then accuracy and cut matrices are saved to appropriate folders in *./dnew/* folder.

I implemented cutting semi-manually, because sometimes you need to try multiple times and implementing that seemed me not rewarding when I compare the time I need to spend. The main missing thing at that was the script were only cutting from the beginning k frames and it was not able to cut from the end. Although most of the cuts needed were at the beginning, there was 2-3 sequences which can be fixed by cutting from the end. But I kept it simple and exclude these sequences from the data-set.

At the end of the script *cleanAndSave* function is called, which basically loads the data from *./dnew/* one more time along with the cut and accuracy matrix and cleans&cuts the data accordingly. Then the cleaned data is saved to *./dclean/* folder. The call graph is below.



You can check the excluded sequences by *itest.animateFalses* script. Or you can manually investigate the excluded sequence by investigating accuracy vectors.

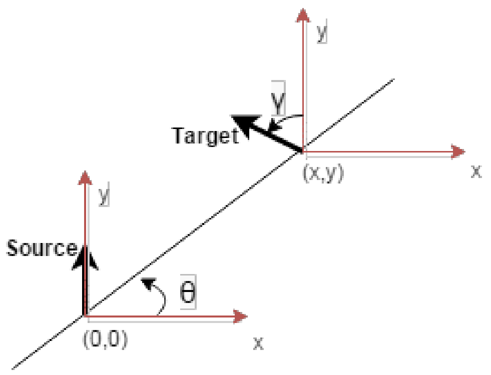
<u>ppro.getStepIndices</u>: This function along with *ppro.getSpeeds* is two important and relatively complex functions. It first combines the Speeds with proper normalization, such that they are quantities from similar interval, such that we can compare them. However one better solution here may normalize the speeds instead of trajectory values. Because, I feel sometimes the direction angle is more dominant then other two since, the direction is able to change (speed) more rapidly.

When I got 3 speed time-series and normalized them; I basically add them and then apply some kind of threshold along with some duration constraints to detect steps. This algorithm works quite well against the variations among subjects. It is able to capture steps with a single threshold and two duration constraint. It gets the middle point of stance duration as step location. This is a drawback when I use the stepping data for body trajectory guessing. Because trajectory is always changing and it must be time sensitive. When I just get the middle point of stationary-interval I lost time information. One improvement may be detecting the beginning of this interval, which corresponds to the touch point of the step.

For not excluding some small steps I've chosen the speed threshold a little smaller. If adjustment steps are not needed one need to apply a filter for that.

3.2. Statistics Extraction & Data Visualization

For investigating the relationship of location and direction deviation with the total number of steps, I have created numerous functions to process the data:



ppro.detectSteps: detects the steps and add relevant fields to the ***seq*** struct.

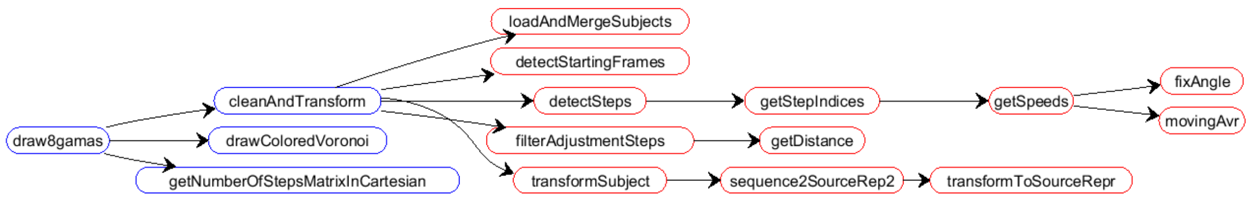
ppro.detectStartingFrames: This function gets a **seq** array and adds starting location and direction to the **seq** struct, which is the average of the starting stance feet. When the object is standing and not moving, this is a valid assumption.

rtools.transformSubject: This function gets a **seq** array and transforms it to the source oriented representation.

ppro.filterAdjustmentSteps: Filters adjustment steps. (needs to be improved)

After the data is processed *plotFun.getNumberOfStepsMatrixInCartesian* function is called which basically prepares 4d matrix with (x,y,gama,totalSteps) and then it is passed to the function *plotFun.drawColoredVoronoi* according to gama intervals manually, which are 8 around 0,45,90… angles.

To get other 3-D plots with (gama, theta, #steps) you can use *plotFun.getNumberOf StepsMatrix* along with *plotFun.drawSurface* and *plotFun.drawSurfaceTri.*

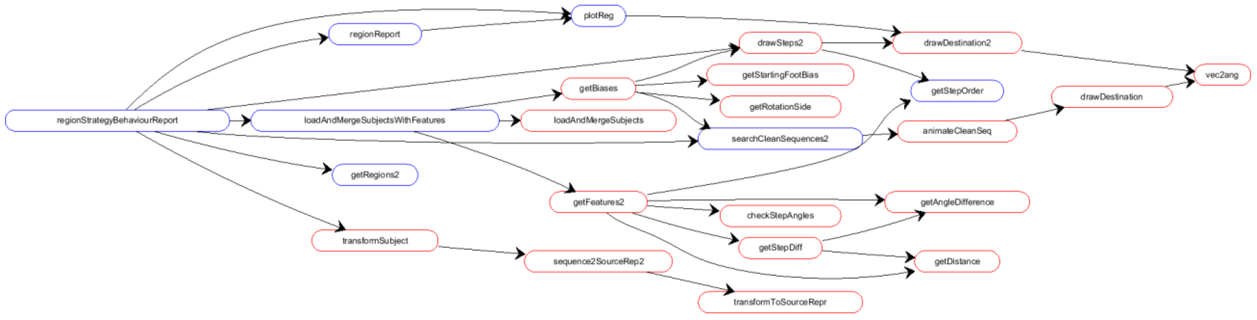


The results of draw8gamas saved in the folder *./gamapics/.*

3.3. Neural Networks Preprocessing

To identify the strategies exists in different regions with different directions I've created a script called *regionStrategyBehaviourReport* in *regtest* package. Which basically save the sequences as Jpegs into the folder *./regionalStrategiesJpegs/* ordered according to the regions with the named convention:

-Region naming: A-B-C
A: Distance [1 5]
B: Theta [1 8]. For example 1 is around 0(around Xaxis) 3 is Y-axis.
C: Gama [1 8]. Target orientation



We the insight I got from *regionStrategyBehaviourReport* results, I've extracted following parameters:

10 features are calculated with ftfun.getFeatures2.m function. 5:8 are used for nnStance and 5:6 is quite corolated with 7:8. [1:4 9:10] total 6 features used at nnNextStep.

%1-ExtraRotationDone
%2-ExtraDistanceTaken
%3-StepRotation-Mean
%4-StepLength
%9 Directness -> is the ratio of the distance taken in the first step to
%the overall distance taken by that foot. Between 0-1
%10 totalSteps;

Strategy Features

%5/6-StartingStance-Width/AngleDeviation
%7/8-EndingStance-Width/AngleDeviation

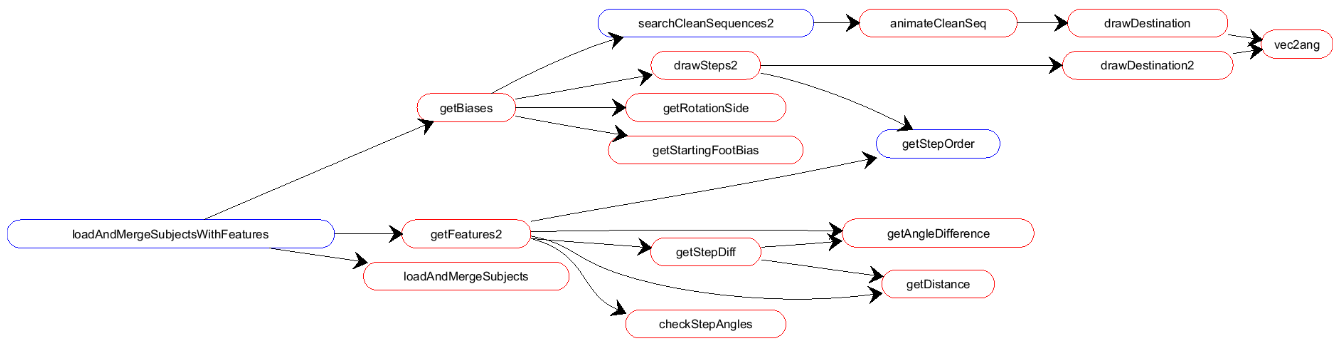
Stance Features

4 Bias is calculated with ftfun.getBiases function to be used in nnFirstStep1 to guess te starting foot.

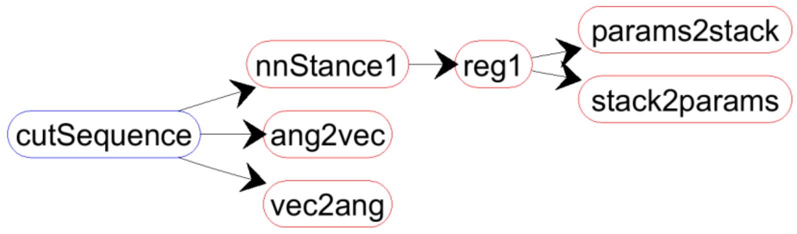
% 1-ForwardStart
% 2-BackwardStart
% 3-ForwardRotationSide

% 4-BackwardRotationSide

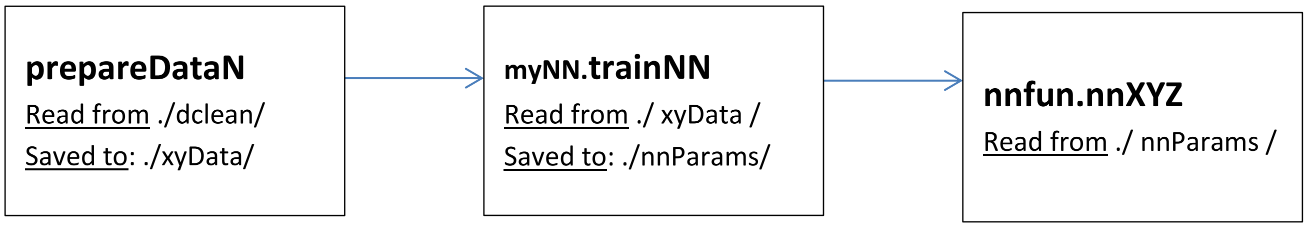
regtest.loadAndMergeWithFeatures function is used for loading cleaning data with feature extraction:



To cut sequences to one step pieces *getseq.cutSequence* function is used. After cutting it also includes one last stationary **cut** which includes the final stance along with a same location next-step. This **cut** is quite important at training to network to converge and stop at the end.



3.4. Neural Networks Training



The template structure I've used at training neural networks is above. First I create a function named as *prepareDataN* where N is a digit. There are several prepareData functions and they are explained below.

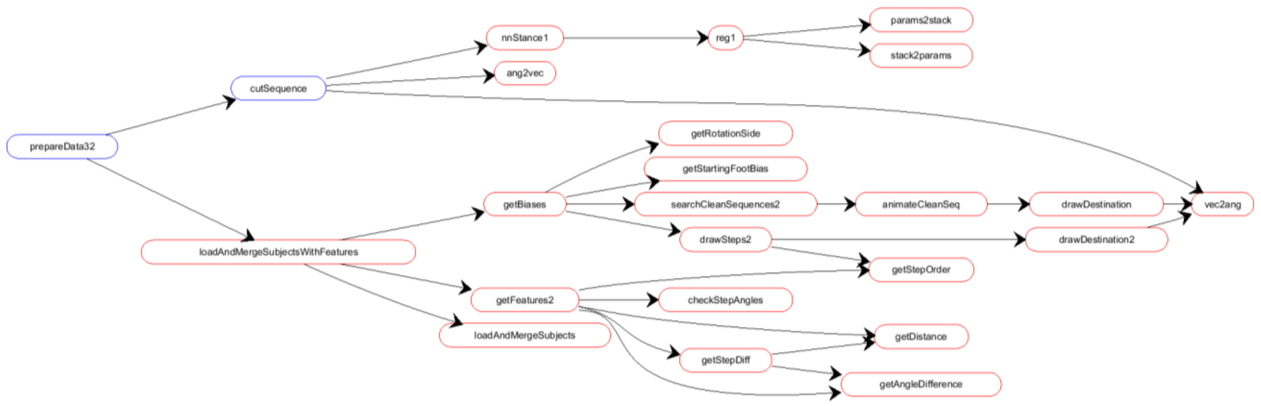
-prepareData1: The first version of next step guessing, not used anymore.

-**prepareData2:** prepares data for *nnStance* which calculates feet parameters given a source in target oriented representation.

-prepareData3/32: prepares data for *nnNextStep*. Second version is used for training.

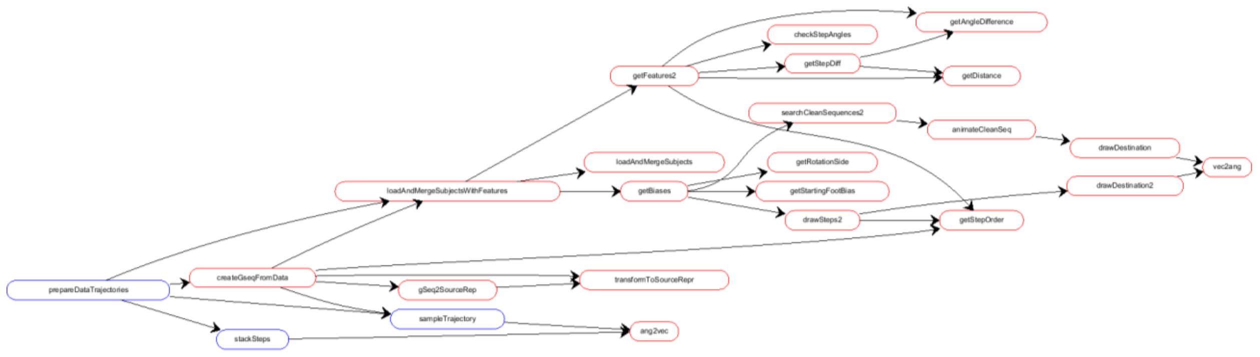
-prepareData4/42: prepares data for first step guessing *nnFirstStep1*, which is RT or LT and this is a classification problem. So *myNN.clsf1* cost function is used at learning. First version is used.

-prepareData5: prepares data for feature set generation, *nnStrategy*. Since there are 1 discrete(one or zero) output value, I've created a new training function, which uses *myNN.comb1* cost function to successfully train the network for mixed output (regression and classification).



- **prepareDataTrajectories:** prepares data for fixed sample trajectory sampling learning, nnTrajectories.

- **prepareDataTrajectories0:** prepares data for fixed sample trajectory sampling learning, nnTrajectories0.



Each data set is saved to ./xyData/ folder and then the name of the dataset is passed to myNN.trainNN function, where the first argument is the name of the data_set and second argument is the name to be saved. Third argument is maximum number of iterations and the last argument is hidden layer structure. There is an optional argument to be 0 if the problem is a classification. For example:

myNN.trainNN('dataTraBdy0','nnTra0Bdy',1900,[400 200 100]);

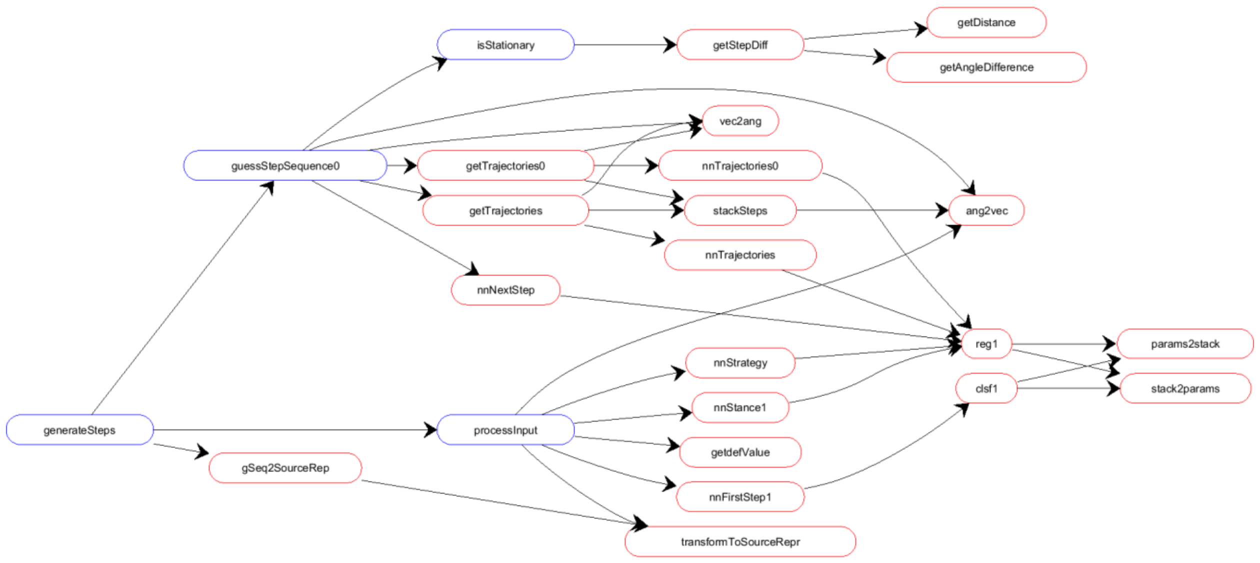
The name of the dataset exists in the end of each prepareData code. After training is done the weights and training parameters are saved to ./nnParams/ folder with the provided save_name. Then a function at nnfun package is crated to normalize the input and forward propagate it. Same cost function used at learning used also here, with forwardPropagation

3.5. Step Sequence & Trajectory Generation

For step sequence and trajectory guessing main function used is *getseq.generateSteps*.

[gSeq,charStruct] = generateSteps(target,charStruct,startingFoot, fixedSamplingFlag)

```
% Possible inputs
        % generateSteps( target,charStruct,startingFoot,fixedSamplingFlag)
        % generateSteps( target,charStruct,startingFoot )
        % generateSteps( target,charStruct)
        % generateSteps( target)
% Inputs:
        % target:           1x3
        % charStruct.bias     4x1
        % charStruct.stance   4x1
        % charStruct.strategy 6x1
        % startingFoot(optional) 1->RT 0->LT
        % fixedSamplingFlag is 1 if you use nnTrajectories.m 0 if you like to use
        % nnTrajectories0.m
% Output:
        % gSeq.steps      (n+2)x3 n=totalSteps; 2 for the starting stance. First
        % step is RT
        % gSeq.stepOrder  1xn
        % gSeq.target    1x3 the starting point for the transform
        % gSeq.bTra (n+1)x4 || gSeq.bTra mx4 It consists of three main functions as seen in the
        diagram.
```



There are 3 main functions called consecutively:

processInput: process target and generate appropriate input for *nnNextStep*. If no charStruct is provided it also guess a feature set and gets default values (*getdefValue*) for bias and stance-features. If no starting foot is provided (-1) it also generates it with *nnFirstStep1* function.

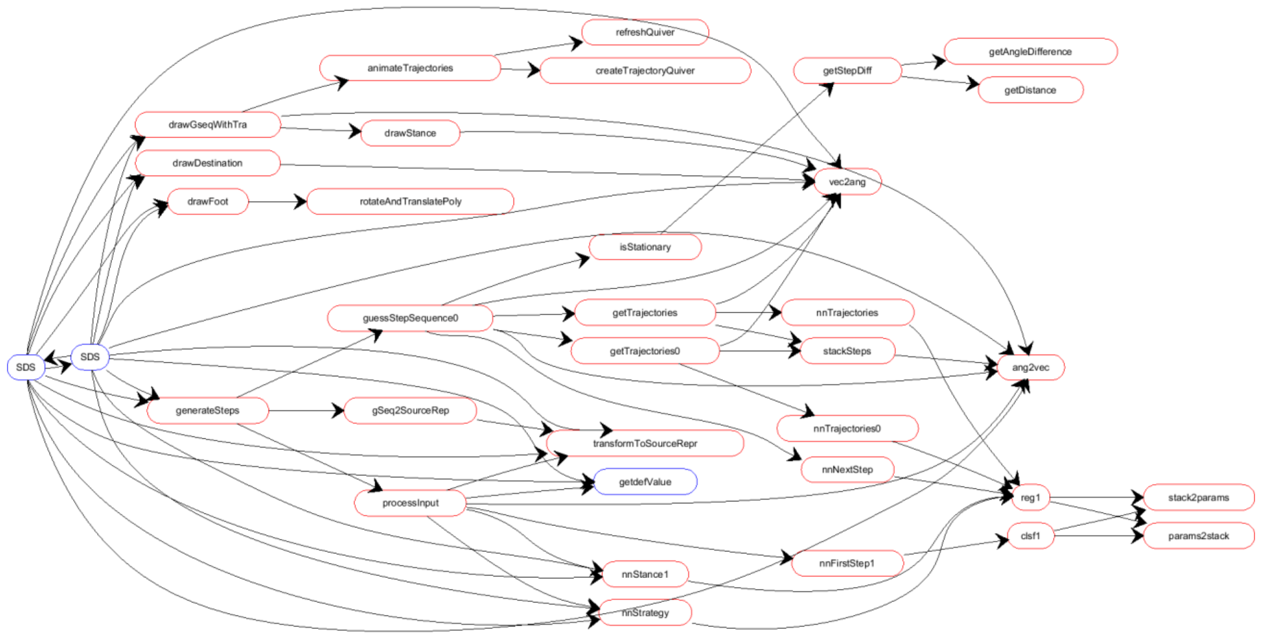
getStepSequence0: generates step sequence with a for loop and stops when *isStationary* function returns 1, which is the case when the difference is small enough for the next step (which means the stepping converged to a stationary position). Then at the end it generates trajectories with one of nnTrajectories functions according to the flag provided.

4. SDS GUI

The GUI is built with the *guide* call in Matlab. It is created to experiment with getseq.genereteSteps function through anim.drawGseqWithTra. The GUI basically shows the generated step sequence and trajectory. The user is able to play with the strategy features, biases and stance features and observe the results. The user can also examine the performance of starting foot guessing and strategy feature generation. The summarized data from the experiment also can be chosen with the Button and text box at the bottom. This data is created with functions: getseq.createGseqFromData/0

The following video demonstrates the usage of the GUI:

-http://youtu.be/KxJEZ1mRKnA



5. DISCUSSION AND FUTURE WORK

Overall, I can say that my program successfully generates realizable and multiple step sequences for a given target. With some modifications and improvements It may become a general model of short distance human stepping. But for me the most important thing that I get from this project is the applicability of Machine Learning algorithms to different fields with appropriate adjustments and I think that I had quite experience with preprocessing, which now allows me to understand neural networks and its training better. I liked to mentioned major difficulties I've encountered during the project and the solutions I've developed:

-Step detection: Normalizing speeds and sum to detect all kinds of steps (rotation, displacement, adjustment)

-Angle representation for neural network input: Each angle is converted to representing unit angle and provided as input.

-Multilayer Hidden Layer and better BackProp: Xaiver's Initialization

-Avoiding over-fitting and generating different strategies: Carefully chosen feature generation

-Trajectory Time Problem: Fixed time sampling and including total #steps as binary vector in the input.

I should also mention the parts which may be improved as much as I remember:

-**Step detection** works well, but I remember that the angle speed is a little dominant to displacement. So a better normalization may be implemented before summing the speeds.

-**Feature extraction** is not bad; it is able to capture different strategies. However It may be improved by selecting them more carefully, but this is hard and I think the features I've extracted are good enough. I should mention character Bias is not useful enough. Because bias features are extracted from a quite small subset of the data (1-5 sequence). These features may be improved by selecting a better subset of data to extract these biases.

-**Neural network architecture and training** is the most important part in the program. With the beginner knowledge I had on ML I've worked with one of the most fundamental things: Neural Networks. But I should say that there may be better algorithms to our problem like RNNs, and Unsupervised Feature Learning methods.

-Another problem with training was I did all the training manually, so the latest parameter sets (network weights) that are used in the nnfuns and in the GUI are the parameters that seemed to me good enough. So I think there is a need for a script or a function to train the network with different hyper parameters and output the best result. I haven't done this since I had short time and I was working on my laptop and had no access to a GPU. But for the best results **automatized and GPU-working analysis** of hyper-parameters are needed and I should also mention that Matlab is not the best language to work with GPU's. There are libraries in the internet for ML with GPU-optimized functions.

-The problem that I encounter rarely when I was playing with GUI was **overlapping feet**, which is caused mostly when the feature set provided with the chosen target are not related enough in terms of data. Which means that combination of features and target or something near does not exist in the training data itself and the neural network failed to learn it appropriately. Instead it tries to guess the steps and since it does not learn the physics of the system it fails to avoid providing overlapping steps. To teach constraints like this one should maybe modify the cost function and add a penalty for that. Unsupervised Feature Learning may also work here. But as I mentioned these cases mostly occurs when random feature sets are chosen. Using guessFeatures in the GUI and using nnStrategy most of the time helps avoiding this problem.

-Another improvement may again **improving cost function** in terms of angle error calculation, which I think might have an impact on the results, especially on trajectory guessing networks. In the cost function while calculating the error signals for back propagation, I've treated each output equally. However this may over-emphasize the angle learning. Because actual error is not caused by the angle vector magnitude, but from the represented angle itself. So the square error function needed to be modified accordingly to capture the angle error and the gradient also be calculated accordingly. So one function for this purpose can be implemented and used for a bigger output like nnTrajectories.

-Although the step generation works well, I can’t say that the **trajectory generation** works as good as step generation. To synchronize the trajectory results with step sequences generated I’ve used the generated step sequences as input to nnTrajectories and because this input is for sure not a training data, the results sometimes fail. **Possible reasons** for this error may include some of the followings:

> *-The recorded data is used at training and it has 44 dimensions and it is hard to generalize. There is a need or **automatized training script** to identify best iteration to stop training.*
>
> *-The recorded sequences and the generated ones look similar. However I don’t think that **getseq.isStationary** function is well-optimized to stop early enough to prevent extra-steps. I have not zero in to this problem, but it seems to me that getseq.isStationary function may be improved generateSteps to produce similar number of steps as output. This would help training nnTrajectory function parameters.*
>
> *-Another idea come up to my mind for better trajectory results is **using directly the data itself** as input instead of using step sequence outputs of the program.*

-After generating the program of step sequence generation I had difficulties with identifying the performance and this also prevented me to write a script to find best network. Because I had only the cost function and the visualization of the output in GUI to decide which network performs better. By looking this two I’ve decided the hyperparamaters of the network. However at this point there is **a metric for identifying the success of a generated stepping sequence** to identify the bad results and optimize further.

-To make a **C++/C library** one need to implement the function generateSteps and its calls. The nnXYZ functions implements forward propagating. So a generic forward propagation function would be enough for all of them and It can be easily found on forums.

5. FUNCTION TABLE

anim.animateCleanSeq.m	anim.drawGseqWithTra.m
anim.animateCuts.m	anim.drawGseqWithTra2.m
anim.animateGivenSequences.m	anim.drawStance.m
anim.animateSteps.m	anim.drawSteps.m
anim.animateTrajectories.m	anim.drawSteps2.m
anim.animateTrajectory.m	anim.drawTra.m
anim.createTrajectoryQuiver.m	anim.refreshQuiver.m
anim.drawDestination.m	
anim.drawDestination2.m	ftfun.checkStepAngles.m
anim.drawFoot.m	ftfun.getAngleDifference.m
anim.drawGseq.m	ftfun.getBiases.m

ftfun.getFeatures1.m
ftfun.getFeatures2.m
ftfun.getRotationSide.m
ftfun.getStartingFootBias.m
ftfun.getStepDiff.m
ftfun.getsAllBiasesInMatrix.m

getseq.createGseqFromData.m
getseq.createGseqFromData0.m
getseq.cutSequence.m
getseq.drawNextStepScript.m
getseq.generateSteps.m
getseq.guessStepSequence0.m
getseq.isStationary.m
getseq.prepareData3.m
getseq.prepareData32.m
getseq.prepareData4.m
getseq.prepareData42.m
getseq.prepareData5.m
getseq.processInput.m
getseq.saveTestSet.m
getseq.testScript.m

itest.animateFalses.m
itest.checkAngles.m
itest.cutAndTest.m
itest.drawMap.m
itest.drawMap2.m
itest.plotSpeedCosts.m
itest.plotSpeedsOneByOne.m
itest.saveTotalSpeedsAsJpeg.m

myNN.clsf1.m
myNN.comb1.m
myNN.grad_check.m
myNN.initialize_weights.m
myNN.load_preprocess_mnist.m
myNN.params2stack.m
myNN.reg1.m
myNN.splitXY.m
myNN.stack2params.m
myNN.trainNN.m
myNN.trainNNmixed.m

nnfun.MatlabNN1.m
nnfun.MatlabNN2.m
nnfun.MatlabNN3.m
nnfun.MatlabNN4.m
nnfun.StopNN.m
nnfun.feedForwardNN.m
nnfun.myNN1.m
nnfun.nnBdyTra2.m
nnfun.nnFirstStep1.m
nnfun.nnNextStep.m
nnfun.nnStance1.m
nnfun.nnStrategy.m
nnfun.nnTrajectories.m
nnfun.nnTrajectories0.m

plotFun.draw8gamas.m
plotFun.drawColoredVoronoi.m
plotFun.drawSurface.m
plotFun.drawSurfaceTri.m
plotFun.getNumberOfStepsMatrix.m
plotFun.getNumberOfStepsMatrixInCartesian.m

ppro.cleanAndSave.m
ppro.cleanAndTransform.m
ppro.cleanSequences.m
ppro.detectStartingFrames.m
ppro.detectSteps.m
ppro.filterAdjustmentSteps.m
ppro.fixAngle.m
ppro.getAllSequences.m
ppro.getDistance.m
ppro.getSpeeds.m
ppro.getStepDistances.m
ppro.getStepIndices.m
ppro.getSubSequence.m
ppro.loadSub.m
ppro.movingAvr.m
ppro.testAccuracyManually.m

ppro2.cutAndFixFrames.m

ppro2.getAngleDiff.m
ppro2.getDistanceAngleError.m
ppro2.getMatrices1.m
ppro2.getMatrices2.m
ppro2.getMatrices3.m
ppro2.loadAndMergeSubjects.m
ppro2.prepareData1.m

pprostance.prepareData2.m
pprostance.visualizeTest.m

regtest.getRegions.m
regtest.getRegions2.m
regtest.getStepOrder.m
regtest.loadAndMergeSubjectsWithFeatures.m
regtest.plotReg.m
regtest.regionReport.m
regtest.regionStrategyBehaviourReport.m
regtest.searchCleanSequences.m
regtest.searchCleanSequences2.m

rtools.ang2vec.m
rtools.gSeq2SourceRep.m
rtools.gSeq2SourceRep2.m
rtools.rotateAndTranslatePoly.m
rtools.sequence2SourceRep.m
rtools.sequence2SourceRep2.m
rtools.transform2polar.m
rtools.transformSubject.m
rtools.transformToSourceRepr.m
rtools.vec2ang.m

tra.getSubTrajectory.m
tra.getTrajectories.m
tra.getTrajectories0.m
tra.prepareDataTrajectories.m
tra.prepareDataTrajectories0.m
tra.sampleTrajectory.m
tra.sampleTrajectory0.m
tra.stackSteps.m