Monoencoder.exe Documentation

Alexander Gilson

Contents

Introduction

No instrument of mankind, no matter how powerful, precise, or expensive, will ever be perfect. The National Synchrotron Light Source II (NSLS-II) is no exception. This massive synchrotron supports around 30 beamlines, one of which is the CSX (Coherent Soft X-Ray Scattering) beamline, which is mostly focused on experiments using x-ray photon correlation spectroscopy (XPCS) and nano-focused scanning diffraction microscopy, or "nanoscattering". However, due to the extreme sensitivity of the instruments at CSX, sources of noise that would be considered irrelevant under most circumstances can cause significant interference. Since there are so many intricate processes operating constantly around and within the beamlines, it can be extremely difficult to find the source of the interference in a reasonable amount of time. However, in comparison to a human, a machine-learning assisted algorithm could greatly expediate that process. Using this algorithm would not only be much faster, but also completely autonomous. This allows it to not only complete its task unsupervised, but also to be replicated an indefinite number of times.

Structure & Dependencies

The program is divided into 4 sections: 'Main', 'Core', 'CNN', and 'Load'. The Main file is where the program is run from, and incorporates all of the necessary elements from the other sections. The Core section is essentially the library of the program, serving as a convenient place to store various methods used throughout the program. The Load section is, as its name may suggest, used to load the input data as well as generate test data that can be used for evaluation and debugging purposes. Finally, the CNN section manages building, saving, and loading the convolutional neural network that plays a key role in the program's functionality.

This program depends on the python libraries Tensorflow, Keras, Numpy, and MatPlotLib in order to run.

Methods

Monoencoder_Core

compare(func_1, func_2, method)

Accepts 2 parameters in the form of functions (func_1 and func_2) and a string called 'method'. It compares the two functions to each other point by point, and outputs a list of values, each between 0 and 1, indicating how strongly the algorithm believes a given pair of points is correlated.

d1d2(point_d1, point_d2)

Accepts two parameters which represent the derivatives at 'point 1' and 'point 2'. Returns a value between 0 and 1 which represents how similar the two parameters are to eachother.

derive(d_func)

Accepts a single parameter in the form of a dictionary generated by beamline_data() which consists of a list of x values and a list of their corresponding y values. It then uses these lists to approximate the derivative of the function described by the two lists of values at any given point p by calculating the slope of the line connecting the points p-1 and p+1, assigning the value of the slope as the derivative at point p. Once it has evaluated the derivative at all points in the function, it will output a new function as a list of x values and a list of y values within a dictionary, only in the outputted version, the y values represent the derivative at their

respective x values instead of the original values of y at their respective points x.

graph(data, f1, f2, count, func_c)

Outputs a graph of the two functions being compared. If there is a group of functions being compared (i.e. more than two) it will output multiple graphs in a grid. Each graph will show two functions: one in grey (f1), and one in color (f2). The color of the points in the color function indicate how likely that function correlates to the function in grey at that particular point with green points representing high correlation, red points representing low correlation, and yellow-ish points represent a moderate correlation value. The optional parameter, func_c, is an optional perameter that helps graph() sort the points that are being compared. It is 'None' by default, and is therefore not required for the method to work.

load_CNN()

Attempts to load an existing neural network model saved to the device. If said model does not exist or can't be found, it will prompt the user to generate one.

match(f_base, f_subject, method)

Accepts two parameters in the form of dictionaries generated by beamline_data(), each consisting of a list of x values and a list of their corresponding y values, and a third parameter which specifies a 'method'. Returns a list of 'pairs' of points, which are to be compared to each other, according to the method that was selected. Acceptable values for 'method' are 'Proximity x', 'Proximity y', 'Interpolation', and 'Latest Update'.

normalize(data)

Accepts a single parameter in the form of a dictionary generated by beamline_data() which consists of a list of x values and a list of their corresponding y values. Returns a 'normalized' version of the input function, with the y values scaled such that the highest y value is represented as 1 the lowest y value is represented as 0, and all other y values are assigned a value that represents their proportionality between the maximum and minimum y values.

p1p2(point_1x, point_1y, point_2x, point_2y, offset)

Accepts 5 parameters, four of which define the x and y values of a pair of points 'point 1' and 'point 2' and an 'offset' parameter used by some comparison methods to compensate for functions that don't begin or end at the same point in time. Outputs a value between 0 and 1 that represents how similar the positions of the points are to each other.

resize(func_j, func_k)

Accepts two parameters in the form of dictionaries generated by beamline_data(), each consisting of a list of x values and a list of their corresponding y values. Converts the lists of x and y values from the two input functions into a 256 x 4 matrix for the program's neural network to analyze.

Monoencoder_CNN

Method_select(neural_net, case)

This method accepts two parameters, one in the form of a neural network model (**NOTE:** ensure a neural network model exists or is generated before this method is called), and the other in the form of a 256 x 4 data matrix outputted by the Monoencoder Core method resize. The function of this method is to use the neural network to select one of the four possible correlation methods used in many of the algorithm's correlation calculations.

Monoencoder_Loader

beamline_data()

A simple method that contains the directories of the files you want to load. Accepts .npy and .npz formats only.

create_function(a, b, c, kind, interval)

Creates a randomly generated function to help train the neural network. Accepts 5 parameters, the first 3 of which are randomly generated floats between -1 and 1, 'kind' which is a string representing the type of function to be generated, and 'interval' which is a random number selected from a specific set (1, 2, 4, 16, 16).

test_data()

This method is used to generate a set of test functions originally used to test the accuracy of the correlation algorithm. As the algorithm has been more or less completed, this function is unused, but left in just in case the correlation algorithm needs to be adjusted or redesigned.

training_data(quantity)

Generates a list of randomly generated functions for training the neural network using the create_function method. The only parameter, quantity, determines the number of functions to be generated.

How to Use

In order to operate the program, you must open Monoencoder_Main, which is set up as the control panel of the program. From there, you can adjust how the program runs in various ways. By default, the program is set to test mode (see line 8: test = True), simply change this to false to have the program load and analyze your data instead. To load your data, open Monoencoder_Load and enter the directory(ies) for the files you want to load under the comment "List files you want to load here" in the beamline_data() method. If you would like to set the minimum threshold for what the algorithm should be considered correlated, on line 40 of Monoencoder_Main, set the value after the '>=' to any number between 0 and 1 (it is set to 0.6 by default). Once all parameters are set to your liking, simply run the repo from Monoencoder_Main and give it a few minutes.