## Neural Network SEBO Documentation

This document highlights the essential functionalities of the code related to the Neural-Network based SEBO.

**Predefined Parameters:**

objectives = {...

* Provides a library of objection functions (2-Dimensional) that consist of the following:
  + The corresponding function written to code to compute response values for physical and simulation observations
  + The function bounds, also known as the design space for which we are searching through.
  + Global maximum point(s) used as reference for generating visuals of the tested function to show how close new points for sampling lie close to the global optimal point(s).
* pretrain\_n: Number of physical observations for SEBO to perform
* train\_sim: Number of simulation experiments for the Neural Networrk to train on during SEBO.
* sim\_stev: Stardard Deviation of simulation response
* max\_bo\_iter: Number of Bayesian Optimization iterations for calibrating simulation model

**Initialization Functions:**

ran\_err = lambda *n*, *x*: r.normal(0, *x*, *n*)

* Adds Gaussian noise to n samples with a standard deviation of x

sim\_err = lambda *x*, *y*, *a*, *b*: 0.1 \* (*a* - 4) \* *x* + 0.05 \* (*b* + 3) \* *y*\*\*2 + 0.03 \* *x* \* *y* + r.normal(0, sim\_stdev)

* Defines the simulation error as a changeable function. The purpose is to introduce bias and noise to simulate a miscalibrated simulator. The function is dependent on:
  + x, y = inputs values of a two-dimensional function
  + a, b = calibration parameters

# Generate n random data points for 2 inputs (X, Y) within specified bounds

X = np.random.uniform(x\_range[0], x\_range[1], pretrain\_n)

Y = np.random.uniform(y\_range[0], y\_range[1], pretrain\_n)

# Calculate the outputs using a predefined benchmark function

output1 = benchmark\_func(X, Y) + ran\_err(pretrain\_n, 0.05)

# Generate Initial Dataset of Physical Experiments

df = pd.DataFrame({'x': X, 'y': Y, 'z': output1})

* Initializes the dataset for pretrian\_n number of physical observations before starting the active learning loop.
* The code returns a data frame of pretrrain\_n generated physical experiments.

sim\_eval = lambda *x*, *y*, *a*, *b*: benchmark(*x*, *y*) + sim\_err(*x*, *y*, *a*, *b*)

* Computes the simulation response of a data point at x and y for a simulator that is currently calibrated at a and b.

def NN\_model(*num\_layers*, *num\_units*, *learning\_rate*, *dropout\_rate*):

* Defines a fully connected feedforward neural network to be used as a surrogate model for simulation predictions with the following configurations:
  + num\_layers: Number of hidden layers
  + num\_units: Number of neurons per layer
  + learning\_rate:
  + dropout\_rate:
  + L2 Regularization: Uses ridge regression to prevent overfitting.
* Ideally, the Neural Network model can be configured however for testing.

def mc\_dropout\_predictions(*model*, *X*, *num\_samples*=50):

* Performs Monte Carlo Dropout by running the given Neural Network model multiple times with dropout enabled at test time. Each forward pass provides a slightly different prediction; therefore, a prediction is approximated over predictions. From this, it is able to produce the mean and standard deviation (uncertainty) from the current model and data passed.
  + model: Defines the neural network model being passed.
  + X: A numpy array of shape (n\_samples, 2) which represents the input features for which we are predicting the simulation response.
  + num\_samples: The number of stochastic forward passes through the neural network with dropout enabled.

**Active Learning Loop:**

def perform\_iteration(*df*):

* This function call performs 1 full cycle of the active learning loop. Each run reads in a data frame of initial physical experiments and outputs the same data frame with pretrain\_n number of added physical experiments from performing SEBO.

def sim\_all(*x*):

…

        return (*df*["diff"]\*\*2).sum()

…

bo = BayesianOptimization(*f*=sim\_all, *domain*=bo\_domain, *acquisition\_type*='EI')

bo.run\_optimization(*max\_iter*=max\_bo\_iter)

* Defines the objective where the squared error between the simulation and physical response is minimized.
* Bayesian optimization is performed to find the optimal a and b parameters over bo\_domain that satisfy the objective.

n\_samples = 100

lhs= qmc.LatinHypercube(*d*=2, *optimization*="random-cd")

lhs\_samples = lhs.random(*n*=train\_sim)

…

scaled\_samples = qmc.scale(lhs\_samples, bounds[:, 0], bounds[:, 1])

* Latin Hypercube Sampling is performed to generate train\_sim additional data points for simulation testing over the given bounds of the current benchmark function.
* “randon-cd” is used as the optimization to ensure an even distribution of points over the entire input space.
* d specifies the dimension of the input space at which points are generated.
* The simulation response is then evaluated for each data point derived from LHS and the complete set of points are then combined with the current initial points with their simulation response into a single data frame sim\_df.

    # Train the model with the given hyperparameters on the entire dataset

    best\_model = NN\_model(

*num\_layers*=6,

*num\_units*=300,

*learning\_rate*=0.0007,

*dropout\_rate*=0.5

    )

# Standardize the Dataset of Simulation Responses

    scaler= StandardScaler().fit(sim\_df[['x', 'y']])

    X\_scaled = scaler.transform(sim\_df[['x', 'y']])

    # Train the best model on the combined standardized dataset

    best\_model.fit(X\_scaled, sim\_df[['sim\_z']], *epochs*=500, *batch\_size*=128, *verbose*=1)

* best\_model defines the architecture of the Neural Network model to fit the simulated samples. The following configurable hyperparameters define the Neural Network model:
  + num\_layers
  + num\_units
  + learning\_rate
  + dropout\_rate
  + epochs
  + batch\_size
* StandardScaler() is used to normalize the inputs of sim\_df before being trained on the neural network surrogate model.

    def p\_nn(*xy*):

*xy* = np.array(*xy*).reshape(1, -1)  # Ensure correct shape (1,2)

        xy\_scaled = scaler.transform(*xy*)

        pred\_mean, pred\_std = mc\_dropout\_predictions(best\_model, xy\_scaled)

        # Check if the point is inside the valid domain

        if (x\_range[0][0] <= *xy*[0, 0] <= x\_range[0][1] and

                y\_range[0][0] <= *xy*[0, 1] <= y\_range[0][1]):

            probability = max(pred\_mean + 1.96 \* pred\_std -sim\_df["sim\_z"].median(), 1e-6)

            return probability

        else:

            return 1e-10

* Defines the probability function for Metropolis\_hastings (MH) sampling
* The input point xy is converted into the shape (1,2) for prediction before being scaled and ran through MC Dropout to obtain its mean and standard deviation.
* The point is then checked to verify that it lies between the valid bounds of the current benchmark function. If it does, a probability-like score is computed using the upper-confidence bound (UCB) minus the median simulated values to center the distribution. A high “score” encourages exploitation of points with high predictions, and exploration of points within uncertain regions. If the point doesn’t lie within the bounds, a near-zero probability is returned.

    def metropolis\_step(*xy*, *sig*):

        prop\_xy = *xy* + np.random.normal(0, *sig*, *size*=*xy*.shape)  # Generate a candidate

        # Ensure proposed point is inside the valid bounds

        if not (x\_range[0][0] <= prop\_xy[0, 0] <= x\_range[0][1] and

                y\_range[0][0] <= prop\_xy[0, 1] <= y\_range[0][1]):

            return *xy*  # Reject out-of-bounds proposals

        u = np.random.uniform(0, 1)

        if u < min(1.0, p\_nn(prop\_xy) / p\_nn(*xy*)):

            return prop\_xy  # Accept new point

        else:

            return *xy*  # Stay at current point

* A candidate point prop\_xy is proposed by adding xy with some Guassian noise of standard deviation sig.
* The function then checks if prop\_xy is within the valid bounds. If it does, then the point is kept.
* prop\_xy is then compared against the current point xy. If the probability score of the new point (prop\_xy) is greater than the current (xy), the new point is accepted otherwise the current point is maintained.

    def mh\_sampler(*sig*, *sample\_size*, *burn\_in*, *lag*):

        # Start at the point with the largest simulated response

        max\_idx = np.argmax(sim\_df["sim\_z"])

        x\_init = sim\_df["x"].iloc[max\_idx]

        y\_init = sim\_df["y"].iloc[max\_idx]

        xy = np.array([[x\_init, y\_init]])

        # Burn-in phase

        for i in range(*burn\_in*):

            xy = metropolis\_step(xy, *sig*)

        # Sampling phase

        output = xy

        for i in range(*sample\_size* - 1):

            for j in range(*lag*):

                xy = metropolis\_step(xy, *sig*)

            output = np.vstack((output, xy))  # Stack valid samples

        return output  # Returns an array of shape (sample\_size, 2)

* The main function for MH Sampling
* The process initially starts with the data point with the highest simulated value. Burn\_in steps are ran to move away from the biased starting point.
* During the sampling phase, lag steps are ran between each sample to reduce autocorrelation while the accepted samples are appended to the list.
* The final output returns sample\_size number of data points for next physical experiments.

# Add points to original dataframe

*df* = pd.concat([*df*, physical\_test\_points], *axis*=0, *ignore\_index*=True)

return *df*

* After the next sampled points for physical experiments are visualized, they are then appended to the original date frame containing all physical sampled points for the pipeline to reiterate again.