研究グループで使うリザバーコンピューティングの要約

2024 年 9 月 15 日

1　リザバーの python コード

$n \in \mathbb{Z}$, $\alpha \in (0, 1]$ に対して、リザバー方程式は以下のように表現される。

$$\begin{aligned} x(n+1) &= (1-\alpha)x(n) + \alpha f(Wx(n) + W_{in}u(n+1)), \\ y(n+1) &:= W_{out}r(n+1), \end{aligned}$$

- $u(n)$ は入力ベクトル、
- $y(n)$ は出力ベクトル、
- $x(n)$ はリザバー状態ベクトル

と言い、

- 入力層のノード数（次元）を N_u,
- リザバーのノード数（次元）を N_x

とする。要は、縦ベクトルとして

$$\begin{aligned} u(n) &= (u_1(n), \cdots, u_{N_u}(n))^T \in \mathbb{R}^{N_u} \\ x(n) &= (x_1(n), \cdots, x_{N_x}(n))^T \in \mathbb{R}^{N_x} \\ y(n) &= (y_1(n), \cdots, y_{N_u}(n))^T \in \mathbb{R}^{N_u} \end{aligned}$$

と表現される。入力ベクトルを高次元空間へ射影したものがリザバー状態ベクトルなので、$N_u \ll N_x$ である。

f を活性化関数と言い、ここでは $f(x) = \tanh x$ と置く。

- W_{in} は $N_x \times N_u$ の入力結合重み行列、
- W は $N_x \times N_x$ のリカレント結合重み行列、

- W_{out} は $N_u \times N_x$ の出力結合重み行列

という。通常のリザバーでは、W_{in} と W は乱数を使って定義する。しかし、この我々研究グループでは、その W_{in} と W の生成に使われている乱数のシード値を、ベイズ最適化によって学習させる。そこが研究として新しい。

1.1 入力結合重み行列の Class

まず、入力結合重み行列 Win を生成するための Class を設定する。Class とは大雑把に言って設計図を意味する。ここではあくまでも Win というリザバー機械学習に対する部品の設計図が書いてあるだけであり、まだ機械学習として実際に実行している段階にはない、というニュアンスである。

乱数に対して乱数シード値（実数値）が一対一に対応している。例えば

$$
\begin{aligned}
&\text{(シード値)} \qquad \text{(乱数)} \\
&\qquad 1 \to 0.23340954 \\
&\qquad 1.1 \to 1.459283 \\
&\qquad 1.2 \to 10.22038894 \\
&\qquad \vdots
\end{aligned}
$$

といった具合である。しかし、この乱数シード値の連続変化が乱数そのものを連続的に変化させているわけではない。実質不連続である。

この研究グループでは optuna（ベイズ最適化）を用いるので、その際、この乱数シード値も学習させることになる。その際、乱数シード値を連続的に変化させたときにそれに対応する行列 W と Win も連続的に変化するように再設定しないといけない。

通常は、乱数シード値（実数値）に対応する乱数が逐次生成されるが、ここでは整数値部分に対応する乱数だけを使用する。まずは、乱数シード値（実数値）を整数部分と小数部分に分離させる。（seed_value が乱数シード値）

```
class Input:
def __init__(self, N_u, N_x, input_scale, seed_value):

seed_int_low = int(np.floor(seed_value))
seed_int_high = int(np.ceil(seed_value))
fraction = seed_value - seed_int_low
```

上のコードは、乱数シード値が与えられたとき、その値にはさまれる整数値を選別している。例えばシード値が seed_value=74.56 であった場合、seed_int_low=74, seed_int_high=75, fraction=0.56 となる。

乱数シードの整数部分を使って乱数行列を生成する。N_x×N_u 行列の各成分に対して、開区間 (-input_scale,input_scale) 内の実数で生成される一様乱数を各成分に代入する。

np.random.seed(seed_int_low)
random_matrix_low = np.random.uniform(-input_scale, input_scale, (N_x, N_u))
np.random.seed(seed_int_high)
random_matrix_high = np.random.uniform(-input_scale, input_scale, (N_x, N_u))

乱数シード値の小数点以下部分を使って、乱数シードの整数部分で生成した乱数行列を以下のように線形補間する。そうすることで、この行列 Win は乱数シード値に対して連続的に変化するものとなる（optuna が使えるようになる）。

interpolated_matrix = (1 - fraction) * random_matrix_low + fraction * random_matrix_high
self.Win = interpolated_matrix

数式としては、N_x×N_u の行列 A,B があったとき、実数値 fraction$\in [0,1]$ を用いて

$$(1-fraction)A + fraction * B$$

という新たな行列を生成しているに過ぎない。例えば、乱数シード値によって行列 A と B が

$$A = \begin{pmatrix} 1 & 3 \\ 0 & 1 \\ 3 & 10 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 2 & 0 \\ 0 & 3 \end{pmatrix}$$

と生成された時、新たな行列は

$$\begin{pmatrix} (1-fraction) & 3*(1-fraction)+fraction \\ 2*fraction & (1-fraction) \\ 3*(1-fraction) & 10*(1-fraction)+3*fraction \end{pmatrix}$$

となる。この新たな行列を採用することで、乱数シード値の連続変化が A から B への連続変化に対応することが一目瞭然である。最後に、この Win を実際に使用する際のコードを以下の call 文で定義する。

```
def __call__(self, u):
return np.dot(self.Win, u)
```

1.2 リザバー（リカレント結合重み行列）の Class

ここではリザバーの Class を定義する。以下は単なる初期化である。

```
class Reservoir:
def __init__(self, N_x, density, rho, activation_func, leaking_rate, seed_value):
self.seed = seed_value
self.W = self.make_connection(N_x, density, rho,seed_value)
self.x = np.zeros(N_x)
self.activation_func = activation_func
self.alpha = leaking_rate
```

この上の make_connection は次で定義しており、リカレント結合重み行列の定義でもある。

```
def make_connection(self, N_x, density, rho,seed_value):
m = int(N_x*(N_x-1)*density/2)
```

この上の m は総結合数、すなわち、行列内のノンゼロとなる行列の成分数を表す。例えば、N_x= 3 で density= 0.4 の場合、m =int(1.2) = 1 となり、

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

といった行列を生成する。

以下、入力結合重み行列の時と同様、乱数シード値を整数部分と小数部分に分離させる。

```
seed_int_low = int(np.floor(seed_value))
seed_int_high = int(np.ceil(seed_value))
fraction = seed_value - seed_int_low
graph_low = nx.gnm_random_graph(N_x, m, seed_int_low)
graph_high = nx.gnm_random_graph(N_x, m, seed_int_high)
edges_low = set(graph_low.edges())
edges_high = set(graph_high.edges())
```

整数の乱数シード値に対応するグラフ（エッジリストという）を生成する。そのグラフ作成のコードが nx.gnm_random_graph となる。ノンゼロを実現させている成分をここではエッジと呼んでいる。分かりやすく述べると、要は、N_x×N_x 行列を生成する際、まずは、各成分に対してゼロかノンゼロどちらを実現させるか、という点をあらかじめ決める、ということである。

例えば、seed_value= 32.56 のとき。seed_int_low= 32 seed_int_high= 33 となり、fraction= 0.56 となる。m= 3 としよう。seed_int_low= 32 の時のグラフが

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

であったとし、seed_int_low= 33 の時のグラフが

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

であったとしよう。この一見バラバラに見える 0 と 1 の配列を、乱数シード値の連続変化に対して（なるべく）連続的に変化するように線形補間するのである。以下、このエッジを補間する（0 か 1 かを選ぶ、というエッジを扱っているので、厳密には連続的にはなり得ない。厳密には不連続だが、なるべく連続性に近いものを生成している、と言ったところである）。まずは、大きい方の整数シード値と小さい方の整数シード値に対応するエッジに対して、共通のエッジと異なるエッジに分ける。

```
common_edges = edges_low & edges_high
only_low_edges = edges_low - edges_high
only_high_edges = edges_high - edges_low
```

以下で、エッジそのものを適切に数量化し、補間に備える。

```
num_only_low_edges = len(only_low_edges)
num_only_high_edges = len(only_high_edges)
num_edges_to_add_from_high = round(fraction * num_only_high_edges)
num_edges_to_remove_from_low =round(fraction * num_only_low_edges)
```

ここで、新しいエッジセットを構築する。list を使って、そのノンゼロとなる成分を [:（ここに上で定義した個数が入る）] を使って足し引きしている状況が想像できれば良い。

```
interpolated_edges = list(common_edges)
```

```
interpolated_edges += list(only_low_edges)[:(num_only_low_edges - num_edges_to_remove_from_low)]
interpolated_edges += list(only_high_edges)[:num_edges_to_add_from_high]
```

ここで、補間された新しいグラフを生成し、それを行列に変換する。
上の例の場合だと、common_edges のグラフが

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

となり、only_low_edges のグラフが

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

only_high_edges のグラフが

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

となる。fraction= 0.56 でやや edges_high に近いので、only_low_edges（44 パーセント）と only_high_edges（56 パーセント）のエッジををこの割合で分割し、それを common_edges に足す、ということである（実際のリザバーでは次元が巨大であり、従って誤差も微小になるが、3×3 行列の場合、なかなか綺麗には割り切れない）。

この段階ではまだゼロかノンゼロかを選別しているだけであり、W の値自体はまだ定まっていない。ここで乱数を使ってその W の各成分の値（ノンゼロとしてあらかじめ定めた成分位置に対して）を適宜定めていく。まず、以下は、グラフを numpy 配列としての行列に変換するコードとなる。

```
interpolated_graph = nx.Graph()
interpolated_graph.add_nodes_from(range(N_x))
interpolated_graph.add_edges_from(interpolated_edges)
interpolated_matrix = nx.to_numpy_array(interpolated_graph)
W = np.asarray(interpolated_matrix)
```

以下、ノンゼロとして指定した成分の値の決め方は、入力結合重み行列の時と同様である。

```
rec_scale = 1.0
seed_int_low = int(np.floor(seed_value))
```

```
seed_int_high = int(np.ceil(seed_value))
fraction = seed_value - seed_int_low
```

整数乱数シード値に対して一様乱数を生成させる

```
np.random.seed(seed_int_low)
random_matrix_low = np.random.uniform(-rec_scale, rec_scale, (N_x, N_x))
np.random.seed(seed_int_high)
random_matrix_high = np.random.uniform(-rec_scale, rec_scale, (N_x, N_x))
```

入力結合重み行列の時と同様の線形補間を行い、それを W にかける

```
interpolated_matrix_value = (1 - fraction) * random_matrix_low + fraction
* random_matrix_high
W *= interpolated_matrix_value
```

行列 W のスペクトル半径を計算し、その指定したスペクトル半径 rho に合わせて W をスケーリングさせる。これは通常のリザバーと同様である。

```
eigv_list = np.linalg.eigh(W)[0]
sp_radius = np.max(np.abs(eigv_list))
W *= rho / sp_radius
return W
```

ここでようやくリザバー方程式が call 文として定義される。

```
def __call__(self, x_in):
self.x = (1.0 - self.alpha) * self.x + self.alpha * self.activation_func(np.dot(self.W,
self.x) + x_in)
return self.x
```

1.3 出力結合重み行列の Class

この class では、出力結合重み行列を定義している。しかし、出力結合重み行列には最適化が本質的に関わっており、かつ、トレーニング段階とテスト段階両方に関係する横断的な概念であるため、この class だけでは、その概念の理解は難しい。

```
class Output:
def __init__(self, N_x, N_y, seed_value):
```

self.Wout = np.zeros((N_y, N_x))
def __call__(self, x):
return np.dot(self.Wout, x)
def setweight(self, Wout_opt):
self.Wout = Wout_opt

上の Wout_opt は、下の最適化の class で生成される。最適化で生成される Wout_opt を Wout に代入しているだけのコードである、ともいえる。

1.4 リッジ回帰の Class

この Class（class 名は Tikhonov）では、最適化（リッジ回帰）を定義している。凸性が使えるので、逆行列を使った明示的な公式を使って Wout を定義している。

class Tikhonov:
def __init__(self, N_x, N_y, beta):
self.beta = beta
self.X_XT = np.zeros((N_x, N_x))
self.D_XT = np.zeros((N_y, N_x))
self.N_x = N_x

下のコマンドでは、d と x に逐次的に代入し、最終的に self.X_XT,self.D_XT を出力させるものとなっている。より具体的には、リザバーを実行させる class（後で定義している）で設定されているループによって、この self.X_XT,self.D_XT がトレーニングデータによって逐次的に更新されることになる。リッジ回帰ではこれは極めて効率的で自然な式変形である。

以下は、後程定義される “optimizer” で実行される。

def __call__(self, d, x):

x = np.reshape(x, (-1, 1))
d = np.reshape(d, (-1, 1))
self.X_XT += np.dot(x, np.transpose(x))
self.D_XT += np.dot(d, np.transpose(x))

以下のコマンドが凸性により得られるよく知られている明示的な公式（最適解）となる。その数式は以下の通り。

$$W_{out} = DX^T(XX^T + \beta I)^{-1}$$

下のコードを見ればわかるが、DX^T が self.D_XT に、XX^T が self.X_XT に対応する。

```
def get_Wout_opt(self):
X_pseudo_inv = np.linalg.inv(self.X_XT + self.beta * np.identity(self.N_x))
Wout_opt = np.dot(self.D_XT, X_pseudo_inv)
return Wout_opt
```

1.5 リザバーを実行させる Class

ここでは、今まで定義した Class を総動員して、リザバーを実行させるための Class を設定している。class 名は ESN である。以下は初期化である。

```
class ESN:
def __init__(self, N_u, N_y, N_x , density, input_scale, rho, activation_func,
leaking_rate, seed_value):
self.seed = seed_value
self.Input = Input(N_u, N_x, input_scale, seed_value)
self.Reservoir = Reservoir(N_x, density, rho, activation_func, leaking_rate,
seed_value)
self.Output = Output(N_x, N_y, seed_value)
self.N_u = N_u
self.N_y = N_y
self.N_x = N_x
```

ここでトレーニングを定義する。そのトレーニングから最適な Wout を選出させる。ここの optimizer は単なる引数としての記号であり、特に意味はない。後で本当にリザバーを実行する際、この引数 optimizer に Tikhonov（の call 文）が代入されていることが分かる。

```
def train(self, U, D, optimizer, trans_len):
train_len = len(U)
for n in range(train_len):
x_in = self.Input(U[n])
```

リザバー状態ベクトルを設定する

```
x = self.Reservoir(x_in)
```

トレーニングする際の教師データ（リザバーの場合は、次の時刻のデータの値）を設定する。ここで出てくる記号 D[n] だけを見てもよく分からないが、リザバーの実行段階においては train_D（トレーニング段階の教師データ）が代入されている。

```
d = D[n]
```

最初の transient time の間はトレーニングさせない。リザバーの基礎概念であるエコーステートプロパティに関係する（が、ここではあまり深く考えなくて良い）。

```
if n > trans_len:
optimizer(d, x)
```

実際、この上の optimizer だけ見ても、その意味はよく分からない。リザバーを実際に実行する段階におけるコードを見た方が分かりやすい。リザバー実行段階に出てくる model.train を見ると、Tikhonov の__call__文を呼び出していることが分かる。すなわち、ここでは、Tiknohov の call 文内の self.X_XT, self.D_XT を逐次更新しているのである。

そして、下のコマンドで、学習済みの出力結合重み行列 Wout を生成している。メソッドを呼び出しているため、() が必要となる。この辺のコードの理解が一番難しい。慣れないうちは「おまじない」程度に捉えておく方がよい。

```
self.Output.setweight(optimizer.get_Wout_opt())
```

上のコマンドによって生成された Wout と、text_len 回繰り返す for 文（漸化式）を使って、予測データの生成を行う。

```
def run(self, U):
test_len = len(U)
Y_pred = []
y = U[0]

for n in range(test_len):
x_in = self.Input(y)
x = self.Reservoir(x_in)
y_pred = self.Output(x)
```

Y_pred.append(y_pred)
y = y_pred
return np.array(Y_pred)

2　リザバーの実行段階：optuna を介する

ここでようやくリザバーの実行段階に突入する。我々研究グループでは optuna を使ってハイパーパラメタも学習（ベイズ最適化として）させる。ここでは、それぞれのパラメタの意味をきちんと理解することが重要となる。

以下は、トレーニングのステップ長さを表す（下の方で、図で表現しているので、それを参考にしながら読み進めるとよい）

T_train = 450

以下は、予測ステップ数を表す。すなわち、このステップ数後の予測データとオリジナルデータを比較する。

T_test = 30

以下は、 一連の「トレーニング・テスト」を行う回数を表す。「トレーニングとテスト」のセットは、一ステップづつ未来方向にずらしていく点を常に意識する（下の図を参照）。

test_num = 5

以下は、optuna でテストする回数を表す

n_trials = 10

以下は、学習しないステップ数、すなわち transient time を表す（実際、あまり気にしなくて良い）

trans_len_D = 10

ここで紹介しているコードは、ベイズ学習で得た結果を csv ファイルとして保存する仕様になっており、非常に便利である。ファイル名は optuna_results.csv である。以下がそのファイルパスとなる。

file_path = 'optuna_results.csv'

⋮
省略
⋮

ベイズ最適化としての目的関数を以下で定義する。ベイズ最適化では、ハイパーパラメタを学習させる。そのハイパーパラメタの取り得る範囲を以下で設定する（上で省略したところの frozen_trial の箇所にも、その取り得る範囲があるので、そこも一致させる）。float は実数で、int は整数を意味する。

```
def objective(trial):

lag = trial.suggest_int("lag", 1, 6)
dim = trial.suggest_int("dim", 1, 9)
N_x = trial.suggest_int("N_x", 40, 300)
beta = trial.suggest_float("beta", 0, 1)
density = trial.suggest_float("density", 0, 1)
input_scale = trial.suggest_float("input_scale", 0, 1)
rho = trial.suggest_float("rho", 0, 1)
alpha = trial.suggest_float("alpha", 0, 1)
seed_value = trial.suggest_float("seed_value", 0, 100)
```

リザバー機械学習では、時系列データを

- トレーニング段階 + 予測に使う初期値（lag∗dim）
- テスト段階

の二つに分ける時刻 T_0 を設定する。

```
T_0 = T_train + lag * dim
```

以下は単なる初期化である。

```
data_delay_train = np.zeros((T_train + 1, dim, test_num), dtype=float)
data_delay = np.zeros((T_train + T_test + 1, dim, test_num), dtype=float)
```

遅れ座標系を使ったリザバー機械学習を適用するため、data_delay_train と

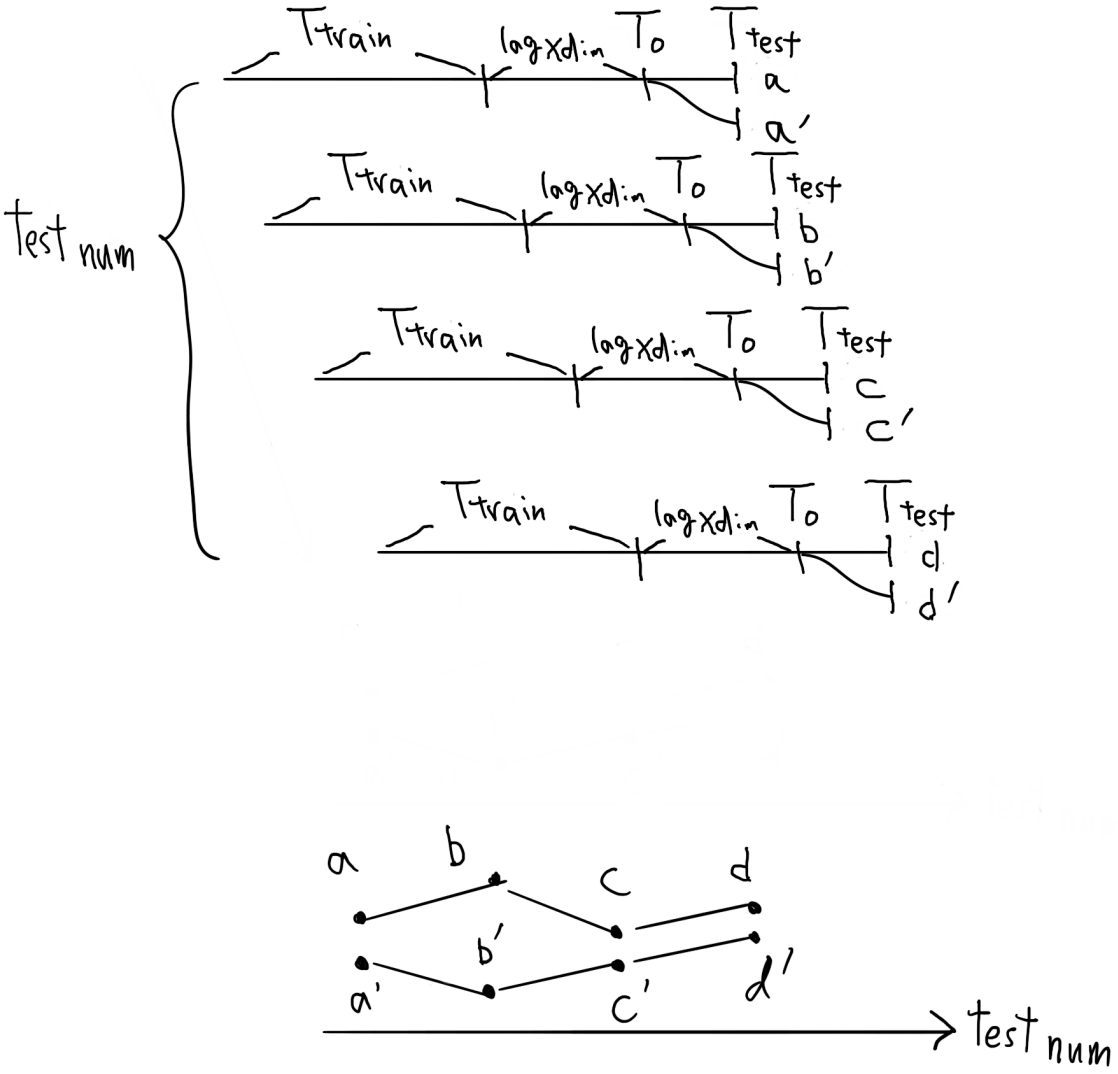

図 1: 上の図は、リザバー機械学習のイメージ図となる。a,b,c,d は T_test ステップ後のオリジナルデータ（フィルタを施したもの）、a',b',c',d' は T_test ステップ後の予測データを意味する。1 ステップずつ未来にズレながら一回一回のリザバー機械学習が行われている様子を図にしている。下の図のようなデータ分布が得られ次第、そこから相関係数（PCC）や RMSE を計算し、モデル性能評価を行う。

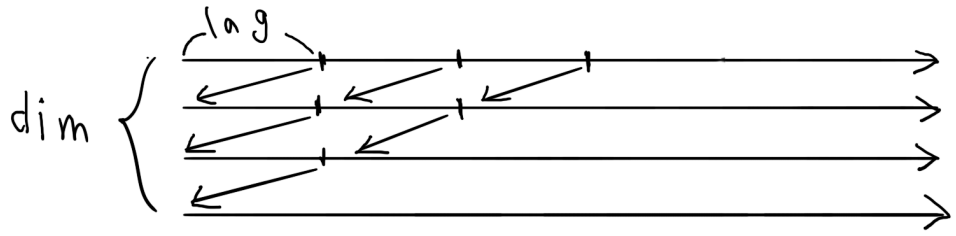

図 2: 上の図は、遅れ座標系を使ったリザバー機械学習のイメージ図となる。一次元時系列データに対して lag 分の遅れを持ったデータを再利用して次元を膨らませる。図では 4 次元の時系列データに膨らませている。

data_delay にデータを格納させる。遅れ座標としての dim と、テストする回数としての test_num の二つのパラメタがある点に注意する（下の図を参照)。

補足 1 なお、下のコマンド内の *data_flt* は、オリジナルデータにノイズ除去のフィルタをかけたものを表す。次章でそのフィルタの説明をする。

```
for t in range(test_num):
for i in range(dim):
data_delay_train[:, i, t] = data_flt[T_0 + t - T_train - i * lag:T_0 + t - i *
lag + 1]
data_delay[:, i, t] = data_flt[T_0 + t - T_train - i * lag:T_0 + t + T_test - i
* lag + 1]
```

以下は単なる初期化である。

```
test_Y = np.zeros((T_test, test_num))
test_D = np.zeros((T_test, test_num))
tentative_test_Y = np.zeros((T_test, dim, test_num))
```

ここで実際のリザバーが実行される。data_delay_train や data_delay に格納したデータをいちいち train_U や train_D に格納して、その格納データに対して model.train が呼び起されリザバーが実行される。optimizer という引数に Tikhonov が代入されている点に注意する。

```
for t in range(test_num):
train_U = np.zeros((T_train, dim), dtype=float)
train_D = np.zeros((T_train, dim), dtype=float)
train_U = data_delay_train[:T_train, :, t]
train_D = data_delay_train[1:T_train + 1, :, t]
model = ESN(train_U.shape[1], train_D.shape[1], N_x, density, input_scale,
rho, activation_func=np.tanh, leaking_rate=alpha, seed_value=seed_value)
train_Y = model.train(train_U, train_D, Tikhonov(N_x, train_D.shape[1],
beta), trans_len=trans_len_D)
tentative_test_Y[:, :, t] = model.run(data_delay[T_train:-1, :, t])
test_Y[:, t] = tentative_test_Y[:, 0, t]
test_D[:, t] = data_delay[T_train + 1:T_train + T_test + 1, 0, t]
```

最後に、このリザバーモデルを評価するために、T_test ステップ後の（test_num 個の）予測データとオリジナルデータに対する RMSE や相関係数（PCC）を取る。ここでは RMSE を取っている。

```
test_Y_cor = [test_Y[T_test - 1, t] for t in range(test_num)]
data_delay_cor = [test_D[T_test - 1, t] for t in range(test_num)]
return RMSE(test_Y_cor, data_delay_cor)
```

以下のコマンドにより、optuna によるベイズ最適化（n_trials 回の試行における）が実行される。

```
study.optimize(objective, n_trials)
    ⋮
  以下省略
    ⋮
```

3　フィルタとデコーダ

得られたオリジナルデータには、そのターゲットとする現象を的確に表すシステム（自励系や、非自励系）とノイズが混在しているものと考えるのが自然である。しかし、実際問題として、そのシステムとノイズを明確に分離することは非常に難しい、そこで、我々研究グループでは、以下の新フィルタ（移動平均の重み関数）によって、ノイズを除去する方法を採用している。

$$weight(t) = g_{r^+,r^-,c} = F_{r^+,r^-}(t)\frac{\sin(\frac{t}{c}-\pi)}{(\frac{t}{c}-\pi)},$$

$$F_{r^+,r^-}(t) = \begin{cases} \dfrac{\sin\left(\frac{t}{r^+}\right)}{t} - \dfrac{\sin\left(\frac{t}{r^-}\right)}{t} & (t \geq 0), \\ 0 \quad (t < 0). \end{cases}$$

ここの c は、移動平均の重み関数の重心を原点に近づけるための調整パラメタである。$[r_-, r_+]$ をパスバンドと呼ぶ。この $weight(t)$ は $t < 0$ でゼロとなっており、未来の情報を一切含めない（未来予測に適した）フィルタになっていることが分かる。ただ、過去に関しては、（数値計算に無限和は存在しないので）適宜カットオフする必要がある。$t > width$ に対して $|weight(t)|$ が十分小さくなる $width$ を選ぶ。そしてオリジナルデータ（ここでは $data$ と置く）に対して

$$data\_flt(t) = \sum_{t'=0}^{width} data(t-t')weight(t') \tag{1}$$

とフィルターを施す。フィルターを施した後のデータを $data\_flt$ と置く。

3.1 デコーダ

我々研究グループでは、フィルタをかけたデータ（トレーニングデータとテストデータ両方）に対してリザバー機械学習モデルを構築している。しかしながら、実際のオリジナルデータには（当たり前のことだが）フィルタがかかっていない。そこで、最終段階として、リザバーモデルから生成される予測データと、実際のオリジナルデータを比較可能にするために（予測データに対して）デコーダをかける。以下がそのデコーダを表す基礎式となる。これは、(1) を直接式変形することで得られるものとなる（data(t) を変数とみなすと漸化式に変貌する点に注意する）。

$$data(t) = \frac{data\_flt(t)}{weight(0)} - \sum_{t'=1}^{width} data(t-t')weight(t')$$

上の式の data_flt を、フィルタデータのリザバー学習で得られた予測データに置き変えて、（data を変数とみなした場合の）漸化式として上の式を走らせる。そうすると、本当に欲しい予測データ、すなわち、従来のオリジナルデータと比較可能な予測データが得られる。

参考文献

[1] 田中剛平，中根了昌，廣瀬明，リザバーコンピューティング，森北出版，2021.