```
'''
This script implements possible data augmentation techniques.

Methods copied from:
B. K. Iwana and S. Uchida, “An empirical survey of data augmentation for time series classification with neural networks,” Plos one, vol. 16, no. 7, p. e0254841, 2021.

Or more specifically from:
https://github.com/uchidalab/time_series_augmentation
'''

import os
import sys
import random
import numpy as np
from tqdm import tqdm
from pathlib import Path
import matplotlib.pyplot as plt

RETURN_PATH = 1
RETURN_VALUE = 0
RETURN_ALL = -1

aug_methods = ["jitter", "scaling", "rotation", "crop", "permutation", "magnitude_warp",
               "time_warp", "window_slice", "spawner", "wdba", "random_guided_warp",
               "random_guided_warp_shape", "discriminative_guided_warp",
"discriminative_guided_warp_shape"]

#------------------------------------------------------------------------------------------------------
```

```
# DTW
#-----------------------------------------------------------------------------------------------------
def _cummulative_matrix(cost, slope_constraint, window):
    p = cost.shape[0]
    s = cost.shape[1]

    # Note: DTW is one larger than cost and the original patterns
    DTW = np.full((p+1, s+1), np.inf)

    DTW[0, 0] = 0.0

    if slope_constraint == "asymmetric":
        for i in range(1, p+1):
            if i <= window+1:
                DTW[i,1] = cost[i-1,0] + min(DTW[i-1,0], DTW[i-1,1])
            for j in range(max(2, i-window), min(s, i+window)+1):
                DTW[i,j] = cost[i-1,j-1] + min(DTW[i-1,j-2], DTW[i-1,j-1], DTW[i-1,j])
    elif slope_constraint == "symmetric":
        for i in range(1, p+1):
            for j in range(max(1, i-window), min(s, i+window)+1):
                DTW[i,j] = cost[i-1,j-1] + min(DTW[i-1,j-1], DTW[i,j-1], DTW[i-1,j])
    else:
        sys.exit("Unknown slope constraint %s"%slope_constraint)

    return DTW

def _traceback(DTW, slope_constraint):
    i, j = np.array(DTW.shape) - 1
    p, q = [i-1], [j-1]
```

```
if slope_constraint == "asymmetric":
    while (i > 1):
        tb = np.argmin((DTW[i-1, j], DTW[i-1, j-1], DTW[i-1, j-2]))

        if (tb == 0):
            i = i - 1
        elif (tb == 1):
            i = i - 1
            j = j - 1
        elif (tb == 2):
            i = i - 1
            j = j - 2

        p.insert(0, i-1)
        q.insert(0, j-1)
elif slope_constraint == "symmetric":
    while (i > 1 or j > 1):
        tb = np.argmin((DTW[i-1, j-1], DTW[i-1, j], DTW[i, j-1]))

        if (tb == 0):
            i = i - 1
            j = j - 1
        elif (tb == 1):
            i = i - 1
        elif (tb == 2):
            j = j - 1

        p.insert(0, i-1)
```

```
        q.insert(0, j-1)
    else:
        sys.exit("Unknown slope constraint %s"%slope_constraint)

    return (np.array(p), np.array(q))

def dtw(prototype, sample, return_flag = RETURN_VALUE, slope_constraint="asymmetric",
window=None):
    """ Computes the DTW of two sequences.
    :param prototype: np array [0..b]
    :param sample: np array [0..t]
    :param extended: bool
    """
    p = prototype.shape[0]
    assert p != 0, "Prototype empty!"
    s = sample.shape[0]
    assert s != 0, "Sample empty!"

    if window is None:
        window = s

    cost = np.full((p, s), np.inf)
    for i in range(p):
        start = max(0, i-window)
        end = min(s, i+window)+1
        cost[i,start:end]=np.linalg.norm(sample[start:end] - prototype[i], axis=1)

    DTW = _cummulative_matrix(cost, slope_constraint, window)
```

```
    if return_flag == RETURN_ALL:
        return DTW[-1,-1], cost, DTW[1:,1:], _traceback(DTW, slope_constraint)
    elif return_flag == RETURN_PATH:
        return _traceback(DTW, slope_constraint)
    else:
        return DTW[-1,-1]

def shape_dtw(prototype, sample, return_flag = RETURN_VALUE, slope_constraint="asymmetric", window=None, descr_ratio=0.05):
    """ Computes the shapeDTW of two sequences.
    :param prototype: np array [0..b]
    :param sample: np array [0..t]
    :param extended: bool
    """
    # shapeDTW
    # https://www.sciencedirect.com/science/article/pii/S0031320317303710

    p = prototype.shape[0]
    assert p != 0, "Prototype empty!"
    s = sample.shape[0]
    assert s != 0, "Sample empty!"

    if window is None:
        window = s

    p_feature_len = np.clip(np.round(p * descr_ratio), 5, 100).astype(int)
    s_feature_len = np.clip(np.round(s * descr_ratio), 5, 100).astype(int)

    # padding
```

```
    p_pad_front = (np.ceil(p_feature_len / 2.)).astype(int)
    p_pad_back = (np.floor(p_feature_len / 2.)).astype(int)
    s_pad_front = (np.ceil(s_feature_len / 2.)).astype(int)
    s_pad_back = (np.floor(s_feature_len / 2.)).astype(int)

    prototype_pad = np.pad(prototype, ((p_pad_front, p_pad_back), (0, 0)), mode="edge")
    sample_pad = np.pad(sample, ((s_pad_front, s_pad_back), (0, 0)), mode="edge")
    p_p = prototype_pad.shape[0]
    s_p = sample_pad.shape[0]

    cost = np.full((p, s), np.inf)
    for i in range(p):
        for j in range(max(0, i-window), min(s, i+window)):
            cost[i, j] = np.linalg.norm(sample_pad[j:j+s_feature_len] - prototype_pad[i:i+p_feature_len])

    DTW = _cummulative_matrix(cost, slope_constraint=slope_constraint, window=window)

    if return_flag == RETURN_ALL:
        return DTW[-1,-1], cost, DTW[1:,1:], _traceback(DTW, slope_constraint)
    elif return_flag == RETURN_PATH:
        return _traceback(DTW, slope_constraint)
    else:
        return DTW[-1,-1]

#----------------------------------------------------------------------------------------------------
# Augmentation functions
#----------------------------------------------------------------------------------------------------

def jitter(x, labels= None, sigma=0.03):
```

```
    # https://arxiv.org/pdf/1706.00527.pdf
    return x + np.random.normal(loc=0., scale=sigma, size=x.shape)

def scaling(x, labels= None, sigma=0.1):
    # https://arxiv.org/pdf/1706.00527.pdf
    factor = np.random.normal(loc=1., scale=sigma, size=(x.shape[0],x.shape[2]))
    return np.multiply(x, factor[:,np.newaxis,:])

def rotation(x, labels= None, ):
    flip = np.random.choice([-1, 1], size=(x.shape[0],x.shape[2]))
    rotate_axis = np.arange(x.shape[2])
    np.random.shuffle(rotate_axis)
    return flip[:,np.newaxis,:] * x[:,:,rotate_axis]

def crop(x, labels= None, sigma= 0.9):
    from scipy.signal import resample

    if sigma > 1:
        raise ValueError("sigma should be <=1")

    if sigma <= 0:
        raise ValueError("sigma should be >0")

    window_size= np.floor(x.shape[1] * sigma).astype(int)
    window_start= np.floor(x.shape[1] * random.uniform(0, 1-sigma)).astype(int)
    window_end= window_start + window_size

    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
```

```
        for dim in range(x.shape[2]):
            cropped = pat[window_start:window_end,dim]
            resampled = resample(cropped, x.shape[1])
            ret[i,:,dim] = resampled
    return ret

def permutation(x, labels= None, max_segments=5, seg_mode="equal"):
    orig_steps = np.arange(x.shape[1])

    num_segs = np.random.randint(1, max_segments, size=(x.shape[0]))

    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
        if num_segs[i] > 1:
            if seg_mode == "random":
                split_points = np.random.choice(x.shape[1]-2, num_segs[i]-1, replace=False)
                split_points.sort()
                splits = np.split(orig_steps, split_points)
            else:
                splits = np.array_split(orig_steps, num_segs[i])
            warp = np.concatenate(np.random.permutation(splits)).ravel()
            ret[i] = pat[warp]
        else:
            ret[i] = pat
    return ret

def magnitude_warp(x, labels= None, sigma=0.2, knot=4):
    from scipy.interpolate import CubicSpline
    orig_steps = np.arange(x.shape[1])
```

```
    random_warps = np.random.normal(loc=1.0, scale=sigma, size=(x.shape[0], knot+2, x.shape[2]))
    warp_steps = (np.ones((x.shape[2],1))*(np.linspace(0, x.shape[1]-1., num=knot+2))).T
    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
        warper = np.array([CubicSpline(warp_steps[:,dim], random_warps[i,:,dim])(orig_steps) for dim in
range(x.shape[2])]).T
        ret[i] = pat * warper

    return ret

def time_warp(x, labels= None, sigma=0.2, knot=4):
    from scipy.interpolate import CubicSpline
    orig_steps = np.arange(x.shape[1])

    random_warps = np.random.normal(loc=1.0, scale=sigma, size=(x.shape[0], knot+2, x.shape[2]))
    warp_steps = (np.ones((x.shape[2],1))*(np.linspace(0, x.shape[1]-1., num=knot+2))).T

    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
        for dim in range(x.shape[2]):
            time_warp = CubicSpline(warp_steps[:,dim], warp_steps[:,dim] *
random_warps[i,:,dim])(orig_steps)
            scale = (x.shape[1]-1)/time_warp[-1]
            ret[i,:,dim] = np.interp(orig_steps, np.clip(scale*time_warp, 0, x.shape[1]-1), pat[:,dim]).T
    return ret

def window_slice(x, labels=None, reduce_ratio=0.9):
    # https://halshs.archives-ouvertes.fr/halshs-01357973/document
```

```
    target_len = np.ceil(reduce_ratio*x.shape[1]).astype(int)
    if target_len >= x.shape[1]:
        return x
    starts = np.random.randint(low=0, high=x.shape[1]-target_len, size=(x.shape[0])).astype(int)
    ends = (target_len + starts).astype(int)

    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
        for dim in range(x.shape[2]):
            ret[i,:,dim] = np.interp(np.linspace(0, target_len, num=x.shape[1]), np.arange(target_len),
pat[starts[i]:ends[i],dim]).T
    return ret

def window_warp(x, labels= None, window_ratio=0.1, scales=[0.5, 2.]):
    # https://halshs.archives-ouvertes.fr/halshs-01357973/document

    #if isinstance(y, (np.ndarray, np.generic) )
    warp_scales = np.random.choice(scales, x.shape[0])
    warp_size = np.ceil(window_ratio*x.shape[1]).astype(int)
    window_steps = np.arange(warp_size)

    window_starts = np.random.randint(low=1, high=x.shape[1]-warp_size-1,
size=(x.shape[0])).astype(int)
    window_ends = (window_starts + warp_size).astype(int)

    ret = np.zeros_like(x)
    for i, pat in enumerate(x):
        for dim in range(x.shape[2]):
            start_seg = pat[:window_starts[i],dim]
```

```
            window_seg = np.interp(np.linspace(0, warp_size-1, num=int(warp_size*warp_scales[i])),
window_steps, pat[window_starts[i]:window_ends[i],dim])
            end_seg = pat[window_ends[i]:,dim]
            warped = np.concatenate((start_seg, window_seg, end_seg))
            ret[i,:,dim] = np.interp(np.arange(x.shape[1]), np.linspace(0, x.shape[1]-1., num=warped.size),
warped).T
    return ret


def spawner(x, labels, sigma=0.05, verbose=-1):
    # https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6983028/
    # use verbose=-1 to turn off warnings
    # use verbose=1 to print out figures

    random_points = np.random.randint(low=1, high=x.shape[1]-1, size=x.shape[0])
    window = np.ceil(x.shape[1] / 10.).astype(int)
    orig_steps = np.arange(x.shape[1])
    l = np.argmax(labels, axis=1) if labels.ndim > 1 else labels

    ret = np.zeros_like(x)
    for i, pat in enumerate(tqdm(x)):
        # guarentees that same one isnt selected
        choices = np.delete(np.arange(x.shape[0]), i)
        # remove ones of different classes
        choices = np.where(l[choices] == l[i])[0]
        if choices.size > 0:
            random_sample = x[np.random.choice(choices)]
            # SPAWNER splits the path into two randomly
            path1 = dtw(pat[:random_points[i]], random_sample[:random_points[i]], RETURN_PATH,
slope_constraint="symmetric", window=window)
```

```
            path2 = dtw(pat[random_points[i]:], random_sample[random_points[i]:], RETURN_PATH,
slope_constraint="symmetric", window=window)
            combined = np.concatenate((np.vstack(path1), np.vstack(path2+random_points[i])), axis=1)
            if verbose:
                print(random_points[i])
            mean = np.mean([pat[combined[0]], random_sample[combined[1]]], axis=0)
            for dim in range(x.shape[2]):
                ret[i,:,dim] = np.interp(orig_steps, np.linspace(0, x.shape[1]-1., num=mean.shape[0]),
mean[:,dim]).T
        else:
            if verbose > -1:
                print("There is only one pattern of class %d, skipping pattern average"%l[i])
            ret[i,:] = pat
    return jitter(ret, labels=labels, sigma=sigma)


def wdba(x, labels, batch_size=6, slope_constraint="symmetric", use_window=True, verbose=-1):
    # https://ieeexplore.ieee.org/document/8215569
    # use verbose = -1 to turn off warnings
    # slope_constraint is for  "symmetric" or "asymmetric"


    if use_window:
        window = np.ceil(x.shape[1] / 10.).astype(int)
    else:
        window = None
    orig_steps = np.arange(x.shape[1])
    l = np.argmax(labels, axis=1) if labels.ndim > 1 else labels


    ret = np.zeros_like(x)
    for i in tqdm(range(ret.shape[0])):
```

```
        # get the same class as i
        choices = np.where(l == l[i])[0]
        if choices.size > 0:
            # pick random intra-class pattern
            k = min(choices.size, batch_size)
            random_prototypes = x[np.random.choice(choices, k, replace=False)]

            # calculate dtw between all
            dtw_matrix = np.zeros((k, k))
            for p, prototype in enumerate(random_prototypes):
                for s, sample in enumerate(random_prototypes):
                    if p == s:
                        dtw_matrix[p, s] = 0.
                    else:
                        dtw_matrix[p, s] = dtw(prototype, sample, RETURN_VALUE,
slope_constraint=slope_constraint, window=window)

            # get medoid
            medoid_id = np.argsort(np.sum(dtw_matrix, axis=1))[0]
            nearest_order = np.argsort(dtw_matrix[medoid_id])
            medoid_pattern = random_prototypes[medoid_id]

            # start weighted DBA
            average_pattern = np.zeros_like(medoid_pattern)
            weighted_sums = np.zeros((medoid_pattern.shape[0]))
            for nid in nearest_order:
                if nid == medoid_id or dtw_matrix[medoid_id, nearest_order[1]] == 0.:
                    average_pattern += medoid_pattern
                    weighted_sums += np.ones_like(weighted_sums)
```

```
            else:
                path = dtw(medoid_pattern, random_prototypes[nid], RETURN_PATH,
slope_constraint=slope_constraint, window=window)
                dtw_value = dtw_matrix[medoid_id, nid]
                warped = random_prototypes[nid, path[1]]
                weight = np.exp(np.log(0.5)*dtw_value/dtw_matrix[medoid_id, nearest_order[1]])
                average_pattern[path[0]] += weight * warped
                weighted_sums[path[0]] += weight

            ret[i,:] = average_pattern / weighted_sums[:,np.newaxis]
        else:
            if verbose > -1:
                print("There is only one pattern of class %d, skipping pattern average"%l[i])
            ret[i,:] = x[i]
    return ret

# Proposed
def random_guided_warp(x, labels, slope_constraint="symmetric", use_window=True,
dtw_type="normal", verbose=-1):
    # use verbose = -1 to turn off warnings
    # slope_constraint is for  "symmetric" or "asymmetric"
    # dtw_type is for shapeDTW or  "normal" or "shape"

    if use_window:
        window = np.ceil(x.shape[1] / 10.).astype(int)
    else:
        window = None
    orig_steps = np.arange(x.shape[1])
    l = np.argmax(labels, axis=1) if labels.ndim > 1 else labels
```

```
    ret = np.zeros_like(x)
    for i, pat in enumerate(tqdm(x)):
        # guarentees that same one isnt selected
        choices = np.delete(np.arange(x.shape[0]), i)
        # remove ones of different classes
        choices = np.where(l[choices] == l[i])[0]
        if choices.size > 0:
            # pick random intra-class pattern
            random_prototype = x[np.random.choice(choices)]

            if dtw_type == "shape":
                path = shape_dtw(random_prototype, pat, RETURN_PATH, slope_constraint=slope_constraint,
window=window)
            else:
                path = dtw(random_prototype, pat, RETURN_PATH, slope_constraint=slope_constraint,
window=window)

            # Time warp
            warped = pat[path[1]]
            for dim in range(x.shape[2]):
                ret[i,:,dim] = np.interp(orig_steps, np.linspace(0, x.shape[1]-1., num=warped.shape[0]),
warped[:,dim]).T
        else:
            if verbose > -1:
                print("There is only one pattern of class %d, skipping timewarping"%l[i])
            ret[i,:] = pat
    return ret

def random_guided_warp_shape(x, labels, slope_constraint="symmetric", use_window=True):
```

```
    return random_guided_warp(x, labels, slope_constraint, use_window, dtw_type="shape")


def discriminative_guided_warp(x, labels, batch_size=6, slope_constraint="symmetric",
use_window=True, dtw_type="normal", use_variable_slice=True, verbose=-1):
    # use verbose = -1 to turn off warnings
    # slope_constraint is for  "symmetric" or "asymmetric"
    # dtw_type is for shapeDTW or  "normal" or "shape"

    if use_window:
        window = np.ceil(x.shape[1] / 10.).astype(int)
    else:
        window = None
    orig_steps = np.arange(x.shape[1])
    l = np.argmax(labels, axis=1) if labels.ndim > 1 else labels

    positive_batch = np.ceil(batch_size / 2).astype(int)
    negative_batch = np.floor(batch_size / 2).astype(int)

    ret = np.zeros_like(x)
    warp_amount = np.zeros(x.shape[0])
    for i, pat in enumerate(tqdm(x)):
        # guarentees that same one isnt selected
        choices = np.delete(np.arange(x.shape[0]), i)

        # remove ones of different classes
        positive = np.where(l[choices] == l[i])[0]
        negative = np.where(l[choices] != l[i])[0]

        if positive.size > 0 and negative.size > 0:
```

```
        pos_k = min(positive.size, positive_batch)
        neg_k = min(negative.size, negative_batch)
        positive_prototypes = x[np.random.choice(positive, pos_k, replace=False)]
        negative_prototypes = x[np.random.choice(negative, neg_k, replace=False)]

        # vector embedding and nearest prototype in one
        pos_aves = np.zeros((pos_k))
        neg_aves = np.zeros((pos_k))
        if dtw_type == "shape":
            for p, pos_prot in enumerate(positive_prototypes):
                for ps, pos_samp in enumerate(positive_prototypes):
                    if p != ps:
                        pos_aves[p] += (1./(pos_k-1.))*shape_dtw(pos_prot, pos_samp, RETURN_VALUE,
slope_constraint=slope_constraint, window=window)
                for ns, neg_samp in enumerate(negative_prototypes):
                    neg_aves[p] += (1./neg_k)*shape_dtw(pos_prot, neg_samp, RETURN_VALUE,
slope_constraint=slope_constraint, window=window)
            selected_id = np.argmax(neg_aves - pos_aves)
            path = shape_dtw(positive_prototypes[selected_id], pat, RETURN_PATH,
slope_constraint=slope_constraint, window=window)
        else:
            for p, pos_prot in enumerate(positive_prototypes):
                for ps, pos_samp in enumerate(positive_prototypes):
                    if p != ps:
                        pos_aves[p] += (1./(pos_k-1.))*dtw(pos_prot, pos_samp, RETURN_VALUE,
slope_constraint=slope_constraint, window=window)
                for ns, neg_samp in enumerate(negative_prototypes):
                    neg_aves[p] += (1./neg_k)*dtw(pos_prot, neg_samp, RETURN_VALUE,
slope_constraint=slope_constraint, window=window)
            selected_id = np.argmax(neg_aves - pos_aves)
```

```
            path = dtw(positive_prototypes[selected_id], pat, RETURN_PATH,
slope_constraint=slope_constraint, window=window)


          # Time warp

          warped = pat[path[1]]

          warp_path_interp = np.interp(orig_steps, np.linspace(0, x.shape[1]-1., num=warped.shape[0]),
path[1])

          warp_amount[i] = np.sum(np.abs(orig_steps-warp_path_interp))

          for dim in range(x.shape[2]):

            ret[i,:,dim] = np.interp(orig_steps, np.linspace(0, x.shape[1]-1., num=warped.shape[0]),
warped[:,dim]).T

        else:

          if verbose > -1:

            print("There is only one pattern of class %d"%l[i])

          ret[i,:] = pat

          warp_amount[i] = 0.

    if use_variable_slice:

      max_warp = np.max(warp_amount)

      if max_warp == 0:

        # unchanged

        ret = window_slice(ret, reduce_ratio=0.9)

      else:

        for i, pat in enumerate(ret):

          # Variable Sllicing

          ret[i] = window_slice(pat[np.newaxis,:,:], reduce_ratio=0.9+0.1*warp_amount[i]/max_warp)[0]

    return ret


def discriminative_guided_warp_shape(x, labels, batch_size=6, slope_constraint="symmetric",
use_window=True):
```

```
    return discriminative_guided_warp(x, labels, batch_size, slope_constraint, use_window,
dtw_type="shape")


#-----------------------------------------------------------------------------------------------------------
# Helper functions to apply on PainData
#-----------------------------------------------------------------------------------------------------------


def get_augmentation(x, y, dataset, method, plot=False):
    """Helper function to retrieve augmented data.
    Either loads the existing data or computes an saves it when augmentation has not been saved before.


    Args:
        x (np): Data.
        y (np): Label.
        dataset (string): Dataset to be used (either 'painmonit' or 'biovid').
        method (fun): The function to augment the dataset.
        plot (bool): Whether to plot the augmentation step or not. Defaults to False.


    Returns:
        np: Augmented data.
    """


    f = Path("datasets", dataset, "aug", method.__name__, "aug.npy")


    if f.exists():
        return np.load(f)


    aug = method(x[:, :, :, 0], labels= y)[..., np.newaxis]
```

```
    if plot:
        random_sample_idx = 0
        sensor_idx = 1
        plt.plot(x[random_sample_idx, :, sensor_idx, 0], label= "Raw")
        plt.plot(aug[random_sample_idx, :, sensor_idx, 0], label= "Augmented")
        plt.title(f"Augmentation: {method.__name__}")
        plt.legend()
        plt.show()

    os.makedirs(f.parent, exist_ok = True)
    np.save(f, aug)

    return aug

def augment(x, y, dataset, l= aug_methods):
    """Function to get augmented data.

    Args:
        x (np): Data.
        y (np): Label.
        dataset (string): Dataset to be used (either 'painmonit' or 'biovid').
        l (list): List of augmentations methods to perform.

    Returns:
        np: Augmented data.
    """

    aug_data = []
```

```
    for method_name in tqdm(l):
        if method_name in aug_methods:
            x_aug = get_augmentation(x.copy(), y, dataset= dataset, method= globals()[method_name])
        else:
            print(f"'{method_name}' is not in the list of implemented methods to augment data!")

        aug_data.append(x_aug)

    x = np.concatenate(aug_data, axis=0)

    return x

if __name__ == "__main__":

    # --- Create augmentation files
    from data_handling import read_biovid_np, read_painmonit_np

    print("Create augmentation for UzL...")
    X_uzl, y_uzl, subjects_uzl = read_painmonit_np(label= "heater")
    augment(X_uzl, y_uzl, dataset= "painmonit")
    print("Augmentation for UzL created.")

    print("Create augmentation for Biovid...")
    X_biovid, y_biovid, subjects_biovid = read_biovid_np()
    augment(X_biovid, y_biovid, dataset= "biovid")
    print("Augmentation for Biovid created.")

    # --- plot example
    print("\nAugmentation example using WDBA")
```

```
x = np.random.rand(5, 2500, 2)

y = np.random.rand(5, 2)

aug = wdba(x, y)


plt.plot(x[0, :, 0], label= "Raw")

plt.plot(aug[0, :, 0], label= "Augmented")

plt.title("WDBA example")

plt.legend()

plt.show()
```