```
# general imports
import os
import math
import random
import numpy as np
import pandas as pd
from pathlib import Path
import matplotlib.pyplot as plt

# machine learning imports
import tensorflow as tf
from keras import layers
import tensorflow.keras.backend as K
from tensorflow.keras.models import Model
from sklearn.feature_selection import RFE
from tensorflow.keras.optimizers import Adam
from sklearn.ensemble import RandomForestRegressor

from .data_handling import unison_shuffled
from .augmentation import window_slice, window_warp

verbose= 0

#----------------------------------------------------------------------------------------------------
# Functions
#----------------------------------------------------------------------------------------------------

def plot_history(history):
```

```
	num_metrics = len(history.history)
	fig, axs = plt.subplots(num_metrics, 1)

	for i, key in enumerate(history.history):
		if num_metrics == 1:
			ax = axs
		else:
			ax = axs[i]
		ax.plot(history.history[key], label= key)
		ax.set_xlabel('epoch')
		ax.legend()

	return fig

#-----------------------------------------------------------------------------------------------------
# Abstract classifier class
#-----------------------------------------------------------------------------------------------------
class classifier(object):
	'''An abstract class to build a network. Here a network receives train_data, test_data, validation_data and parameters.
	Functions for data proccessing and model evaluation can be overwritten. A function to create a model has to be overwritten and must return a keras model.

	During init the functions "data_processing" and "create_model" are called.

	Usage example
	----------

	network = cnn(param)
```

```
	network.set_dataset(train_data= (x_train, y_train), test_data= (x_test, y_test), aug_data= 
(aug_train, aug_test),
						hcf_data= (hcf_train, hcf_test), sub_data= (sub_train, 
sub_test))

	network.data_processing()
	network.create_model()

	network.train()

	prediction = network.predict_x_test()

	Functions
	----------

	data_processing(): Can be overwritten.

	create_model(): Function to implement a (keras) model. Must return a (keras) model that has a 
function 'predict'.

	train(): Function to train the model.
	'''

	def __init__(self, param = None, name= None):
		self.name = name
		if param is not None:
			self.set_param(param)

		self.callbacks = None
```

```
def __str__(self):
    return self.name

def set_param(self, param):
    self.param = param

def set_dataset(self, train_data, test_data, hcf_data, sub_data, aug_data):

    self.x_train = train_data[0]
    self.y_train = train_data[1]
    self.x_test = test_data[0]
    self.y_test = test_data[1]
    self.hcf_train = hcf_data[0]
    self.hcf_test = hcf_data[1]
    self.sub_train = sub_data[0]
    self.sub_test = sub_data[1]
    self.aug_train = aug_data[0]
    self.aug_test = aug_data[1]
    assert self.x_train.shape[1:] == self.x_test.shape[1:]

    self.num_classes = self.y_train.shape[1]
    self.num_sensors = self.x_train.shape[2]

def data_processing(self):
    # If there is augmentation data unequal to raw data and param is said, we extend 'train'
with 'aug train'
    if self.aug_train is not None and not np.array_equal(self.x_train, self.aug_train) and
"aug" in self.param and len(self.param["aug"]) > 0:
```

```
                        self.x_train = np.concatenate([self.x_train, self.aug_train], axis= 0)
                        self.y_train = np.concatenate([self.y_train] + [self.y_train] *
len(self.param["aug"]), axis=0)
                        self.sub_train = np.concatenate([self.sub_train] + [self.sub_train] *
len(self.param["aug"]), axis=0)

                        # If the param is given, we extend the 'test' set with the augmented test set
                        if "use_aug_test" in self.param and self.param["use_aug_test"]:
                                self.x_test = np.concatenate([self.x_test, self.aug_test], axis= 0)
                                self.y_test = np.concatenate([self.y_test] + [self.y_test] *
len(self.param["aug"]), axis=0)
                                self.sub_test = np.concatenate([self.sub_test] + ([self.sub_test] *
len(self.param["aug"])), axis=0)

        def return_dataset(self):
                """Function to retrieve the used dataset of the network.

                Returns
                -------
                np: Dataset. (x_train, y_train), (x_test, y_test)
                """
                return (self.x_train, self.y_train), (self.x_test, self.y_test)

        def create_model(self):
                '''
                Abstract class. Has to be implemented. Should return evaluation score and a model.
                '''
                pass

        def train(self, plot_info = False):
```

```
        """Function to fit the classifier with training data and predict the outcome on test data.
        Return the accuracy, sensivity and specifity in a list.

        Parameters
        ----------
        plot_info: Bool. Whether to plot the training history or not. Default is False.
        """
        param = self.param

        self.model.compile(loss= "categorical_crossentropy", optimizer=
Adam(learning_rate=param["lr"]), metrics= ["accuracy"])

        history = self.model.fit(self.x_train, self.y_train, epochs= param["epochs"], verbose=
verbose, batch_size= param["bs"],
            validation_data = (self.x_test, self.y_test), callbacks = self.callbacks)

        if plot_info:
            plot_history(history.history)



    def predict_test(self):
        """Function to return the prediction for the test data of the model.

        Calls model.predict(self.x_test, verbose=verbose)

        Returns
        -------
        np: prediction
        """
```

```
        return self.model.predict(self.x_test, verbose= verbose)

    def predict_train(self):
        """Function to return the prediction for the test data of the model.

        Calls model.predict(self.x_test, verbose=verbose)

        Returns
        -------
        np: prediction
        """
        return self.model.predict(self.x_train, verbose= verbose)

    def get_feature_layer(self):
        """Function to return the feature layer, if it exists.

        Returns
        -------
        layer
        """

        # check if the instance has a `model` attribute - if not then raise an Error
        if not hasattr(self, 'model'):
            raise ValueError("Instance has no attribute 'model'.")

        # get layers with matching names
        matching_layers = [layer for layer in self.model.layers if layer.name ==
self.feature_layer_name]
```

```
        # raise an error if we do not find 1 matching layer
        if len(matching_layers) != 1:
            raise ValueError("""Wrong number of feature layers found.
                                                Should be 1, but is
'{}'.""".format(len(matching_layers)))

        return matching_layers[0]

    def output_mlp(self, inputs):
        """Apply a common output MLP with a feature layer.

        Returns
        -------
        layer
        """

        if type(inputs) == list:
            inputs = layers.concatenate(inputs)

        x = layers.Flatten()(inputs)

        self.feature_layer_name = "all_features"

        x = layers.Dense(100, activation= "relu", name= self.feature_layer_name)(x)

        x = layers.Dense(self.param["dense_out"], activation= "relu")(x)

        x = layers.Dense(self.num_classes, activation='softmax')(x)
```

```
        return x

    def get_features(self, subject):
        if self.features_available(subject):
            return self.load_features(subject)

        self.create_model()
        self.train()

        # remove classification layer
        model = Model(self.model.inputs, self.model.layers[-2].output)

        # transform data from  raw input to DL features
        [features_train, features_test] = [model.predict(x, verbose= verbose) for x in
[self.x_train, self.x_test]]

        # tranform numpy to pandas
        features_train = pd.DataFrame(features_train, columns = [f"DL_{i}" for i in
range(features_train.shape[1])])
        features_test = pd.DataFrame(features_test, columns = [f"DL_{i}" for i in
range(features_test.shape[1])])

        self.save_features(features_train, features_test, subject)

        return features_train, features_test

    def feature_path(self, subject):
        return Path("feature", self.param["dataset"], self.name, str(subject))

    def features_available(self, subject):
```

```
			return Path(self.feature_path(subject), "train.csv").exists() and 
Path(self.feature_path(subject), "test.csv").exists()


		def load_features(self, subject):
			train =  pd.read_csv(Path(self.feature_path(subject), "train.csv"), sep=";", decimal=",")
			test =  pd.read_csv(Path(self.feature_path(subject), "test.csv"), sep=";", decimal=",")
			return train, test


		def save_features(self, train, test, subject):
			os.makedirs(self.feature_path(subject), exist_ok = True)
			train.to_csv(Path(self.feature_path(subject), "train.csv"), sep= ";", decimal= ",", 
index=False)
			test.to_csv(Path(self.feature_path(subject), "test.csv"), sep= ";", decimal= ",", 
index=False)


class rf(classifier):
		'''
		A wrapper class to train a random forest using a Keras classifier.
		Uses self.model, removes the last layer and tranforms the train/test set to train and evaluate a 
random forest.
		'''
		def __init__(self, param, name = "rf"):
			super().__init__(param, name)


			# default param
			default_param = {'n_estimators': 100, 'max_depth': None, 'min_samples_split': 2}
			self.param = default_param
			self.param.update(param)


		def train(self):
```

```
            # RFE
            if "rfe_manual" in self.param and self.param["rfe_manual"] != None and type(self.param["rfe_manual"]) == int:
                model = RandomForestRegressor(n_estimators = self.param["n_estimators"], max_depth= self.param["max_depth"], min_samples_split= self.param["min_samples_split"])
                selector = RFE(model, n_features_to_select= self.param["rfe_manual"], step=self.param["step"])
                selector = selector.fit(self.hcf_train, self.y_train)
                best_features = list(self.hcf_train.columns[selector.support_])
                self.hcf_train = self.hcf_train[best_features]
                self.hcf_test  = self.hcf_test[best_features]
                self.param["rfe_features"]= best_features

            # Train the model on training data
            self.model = RandomForestRegressor(n_estimators = self.param["n_estimators"], max_depth= self.param["max_depth"], min_samples_split= self.param["min_samples_split"])
            [self.hcf_train, self.y_train] = unison_shuffled([self.hcf_train, self.y_train])
            self.model.fit(self.hcf_train, self.y_train)

    def predict_test(self):
        return self.model.predict(self.hcf_test)

    def get_features(self, subject):
        return self.hcf_train, self.hcf_test

class rf_wrapper(classifier):
    '''
    A wrapper class to train a random forest using a Keras classifier.
```

```
        Uses self.model, removes the last layer and tranforms the train/test set to train and evaluate a random forest.
        '''
        def train(self):
                # train keras model
                super(rf_wrapper, self).train()

                # remove classification layer
                model = Model(self.model.inputs, self.model.layers[-2].output)

                # transform data from  raw input to DL features
                [features_train, features_test] = [model.predict(x, verbose= verbose) for x in [self.x_train, self.x_test]]

                # tranform numpy to pandas
                self.hcf_train = pd.DataFrame(features_train, columns = [f"DL_{i}" for i in range(features_train.shape[1])])
                self.hcf_test  = pd.DataFrame(features_test,  columns = [f"DL_{i}" for i in range(features_test.shape[1])])

                # Instantiate model with 100 decision trees
                self.model = RandomForestRegressor(n_estimators = 100)
                # Train the model on training data
                [self.hcf_train, self.y_train] = unison_shuffled([self.hcf_train, self.y_train])
                self.model.fit(self.hcf_train, self.y_train)

        def predict_test(self):
                """Function to return the prediction for the test data of the model.

                Calls model.predict(self.x_test)
```

```
        Returns
        -------
        np: prediction
        """
        return self.model.predict(self.hcf_test)

#----------------------------------------------------------------------------------------------------
# MLP
#----------------------------------------------------------------------------------------------------
class mlp(classifier):
    def __init__(self, param, name = "mlp"):
        super(mlp, self).__init__(param, name)

    def create_model(self):
        input_layer = layers.Input(self.x_test.shape[1:])

        flat = layers.Flatten()(input_layer)

        #--- Encoder
        x = layers.Dropout(0.2)(flat)
        x = layers.Dense(250, activation= "relu")(x)
        x = layers.Dropout(0.2)(x)
        x = layers.Dense(100, activation= "relu")(x)
        x = layers.Dropout(0.2)(x)

        out = layers.Dense(self.num_classes, activation='softmax')(x)

        self.model = Model(inputs= input_layer, outputs= out)
```

```
class mlp_rf(mlp, rf_wrapper):
    def __init__(self, param, name = "mlp_rf"):
        super(mlp_rf, self).__init__(param, name)


#-----------------------------------------------------------------------------------------------------
# CNN
#-----------------------------------------------------------------------------------------------------


class cnn(classifier):
    def __init__(self, param, name = "cnn"):
        super().__init__(param, name)


    def create_model(self):
        input_layer = layers.Input(self.x_test.shape[1:])


        if "norm_layer" in self.param and self.param["norm_layer"]:
            x = layers.LayerNormalization(axis= 1)(input_layer)
        else:
            x = input_layer


        x = layers.Conv2D(128, (7, 1), activation= "relu", strides= (2, 1))(x)
        x = layers.MaxPool2D((4, 1))(x)
        x = layers.Dropout(0.2)(x)


        x = layers.Conv2D(64, (11, 1), activation= "relu", strides= (2, 1))(x)
        x = layers.MaxPool2D((4, 1))(x)
        x = layers.Dropout(0.2)(x)
```

```
        x = layers.Conv2D(32, (7, 1), activation= "relu", strides= (2, 1))(x)
        x = layers.MaxPool2D((4, 1))(x)
        x = layers.Dropout(0.2)(x)

        out = self.output_mlp(x)

        self.model = Model(inputs= input_layer, outputs= out)

class cnn_rf(cnn, rf_wrapper):
    def __init__(self, param, name = "cnn_rf"):
        super(cnn_rf, self).__init__(param, name)

#---------------------------------------------------------------------------------------------------
# Autoencoder wrapper
#---------------------------------------------------------------------------------------------------

class autoencoder_wrapper(classifier):
    '''
    This class implements an autoencoder wrapper.
    To implement a AE network, use this wrapper and implement a "create_ae()" function.
    '''

    @staticmethod
    def closest_power(x):
        '''
        Function to calculate the nearest smaller "power of two" (2^n) to x.
        '''
        n = math.floor(math.log(x, 2))
        return pow(2, n)
```

```
    def __init__(self, param, name = "autoencoder_wrapper"):
        super(autoencoder_wrapper, self).__init__(param, name)

    def create_model(self):
        '''
        Function to create the keras model.
        Parameters
        ----------

        Returns
        -------
        Keras model: Sequential keras model with 3 blocks.
        '''

        # Create AE
        self.create_ae()
        self.ae.compile(loss= "mse", optimizer= Adam(learning_rate= self.param["lr"]), metrics=
["mae"])
        self.ae.fit(self.x_train, self.x_train, epochs= self.param["epochs"], verbose= verbose,
batch_size= 8,
                                validation_data = (self.x_test, self.x_test))

        # freeze encoder part
        for layer in self.encoder.layers:
            layer.trainable=False

        #--- Classifier
        flat = layers.Flatten()(self.encoder.output)
```

```
        d = layers.Dense(100)(flat)
        d = layers.Dense(50)(d)
        d = layers.Dense(25)(d)

        num_classes = self.y_train.shape[1]
        out = layers.Dense(num_classes, activation='softmax')(d)

        self.model = Model(inputs = self.encoder.input, outputs = out)

    def train(self):
        """Function to fit the classifier with training data and predict the outcome on test data.
        Return the accuracy, sensivity and specifity in a list.

        Parameters
        ----------
        model: Keras model.

        Returns
        -------
        int: accuracy
        keras model
        """

        self.model.compile(loss= 'categorical_crossentropy', optimizer=
Adam(learning_rate=0.0001), metrics= ["accuracy"])

        self.model.fit(self.x_train, self.y_train, epochs= self.param["epochs"], verbose= verbose,
batch_size= self.param["bs"],
                validation_data = (self.x_test, self.y_test))
```

```
        result = self.model.evaluate(self.x_test, self.y_test, verbose=0)

        return result[self.model.metrics_names.index("accuracy")], self.model

    def predict_test(self, verbose= verbose, plot= False):
        """Function to return the prediction for the test data of the model.

        Calls model.predict(self.x_test, verbose=verbose)

        Parameters
        ----------
        Verbose: int. Verbose mode of the keras function.

        Returns
        -------
        np: prediction
        """

        if plot:
            num_sensors = self.x_test.shape[2]
            for i, data in enumerate(self.x_test):
                original_data = data[np.newaxis, ...]
                ae_data = self.ae.predict(original_data)

                for sensor_id in np.arange(num_sensors):
                    fig, axs = plt.subplots(2, 1)
                    raw_input = original_data[0, :, sensor_id, 0]
                    axs[0].plot(raw_input)
```

```
                    axs[0].set_title('Raw sensor data')
                    axs[1].plot(ae_data[0, :, sensor_id, 0])
                    axs[1].set_title('AE output')
                    fig.suptitle("Autoencoder on 'x_test'.\nSensor: {}\nLabel: {}".format(self.param["selected_sensors"][sensor_id], self.y_test[i]), fontsize=16)
                    plt.show()

        return self.model.predict(self.x_test, verbose=verbose)


#---------------------------------------------------------------------------------------------------
# Convolutional Autoencoder + MLP
#---------------------------------------------------------------------------------------------------
class cae(autoencoder_wrapper):
    '''
    Implements a convolutional autoencoder classifier porposed by Mellado et al.
    (https://www.researchgate.net/publication/320970841_Pseudorehearsal_Approach_for_Incremental_Learning_of_Deep_Convolutional_Neural_Networks/link/5a709537a6fdcc33daa9c821/download)


    The autoencoder and classification part of the neural network is trained in parallel.
    '''


    def __init__(self, param, name = "cae"):
        super(cae, self).__init__(param, name)


    def create_ae(self):
        def down_scale_block(inputs, filters, kernel, pool):
            conv = layers.Conv2D(filters= filters, kernel_size= kernel, activation = 'relu', padding = 'same')(inputs)
            result = layers.MaxPooling2D(pool_size= pool)(conv)
```

```
            return result

        def up_scale_block(inputs, filters, kernel, pool):
            up = layers.UpSampling2D(size = pool)(inputs)
            result = layers.Conv2D(filters= filters, kernel_size= kernel, activation = 'relu',
padding = 'same')(up)
            return result

        inputs = layers.Input(self.x_test.shape[1:])

        #--- Encoder
        down = down_scale_block(inputs, filters= 64, kernel= (7, 1), pool= (4, 1))
        down = down_scale_block(down, filters= 32, kernel= (11, 1), pool= (4, 1))
        encoded = down_scale_block(down, filters= 16, kernel= (11, 1), pool= (4, 1))

        #--- Decoder
        up = up_scale_block(encoded, filters= 16, kernel= (11, 1), pool= (4, 1))
        up = up_scale_block(up, filters= 32, kernel= (11, 1), pool= (4, 1))
        n = up_scale_block(up, filters= 64, kernel= (7, 1), pool= (4, 1))

        out = layers.Conv2D(1, (1, 1), padding='same')(n)

        self.encoder = Model(inputs = inputs, outputs = encoded)
        self.ae = Model(inputs = self.encoder.input, outputs = out)

#----------------------------------------------------------------------------------------------------
# Convolutional Autoencoder + RF
#----------------------------------------------------------------------------------------------------
class cae_rf(cae, rf_wrapper):
```

```
        def __init__(self, param, name = "cae_rf"):
                super(cae_rf, self).__init__(param, name)

#-----------------------------------------------------------------------------------------------------
# Supervised contrastive CAE
#-----------------------------------------------------------------------------------------------------

def npairs_loss(y_true, y_pred) -> tf.Tensor:
    """Computes the npairs loss between `y_true` and `y_pred`.

    Npairs loss expects paired data where a pair is composed of samples from
    the same labels and each pairs in the minibatch have different labels.
    The loss takes each row of the pair-wise similarity matrix, `y_pred`,
    as logits and the remapped multi-class labels, `y_true`, as labels.

    The similarity matrix `y_pred` between two embedding matrices `a` and `b`
    with shape `[batch_size, hidden_size]` can be computed as follows:

    >>> a = tf.constant([[1, 2],
    ...                 [3, 4],
    ...                 [5, 6]], dtype=tf.float16)
    >>> b = tf.constant([[5, 9],
    ...                 [3, 6],
    ...                 [1, 8]], dtype=tf.float16)
    >>> y_pred = tf.matmul(a, b, transpose_a=False, transpose_b=True)
    >>> y_pred
    <tf.Tensor: shape=(3, 3), dtype=float16, numpy=
    array([[23., 15., 17.],
       [51., 33., 35.],
```

```
    [79., 51., 53.]], dtype=float16)>

  <... Note: constants a & b have been used purely for
  example purposes and have no significant value ...>

  See: http://www.nec-labs.com/uploads/images/Department-Images/MediaAnalytics/papers/nips16_npairmetriclearning.pdf

  Args:
    y_true: 1-D integer `Tensor` with shape `[batch_size]` of
      multi-class labels.
    y_pred: 2-D float `Tensor` with shape `[batch_size, batch_size]` of
      similarity matrix between embedding matrices.

  Returns:
    npairs_loss: float scalar.
  """
  y_pred = tf.convert_to_tensor(y_pred)
  y_true = tf.cast(y_true, y_pred.dtype)

  # Expand to [batch_size, 1]
  y_true = tf.expand_dims(y_true, -1)
  y_true = tf.cast(tf.equal(y_true, tf.transpose(y_true)), y_pred.dtype)
  y_true /= tf.math.reduce_sum(y_true, 1, keepdims=True)

  loss = tf.nn.softmax_cross_entropy_with_logits(logits=y_pred, labels=y_true)

  return tf.math.reduce_mean(loss)
```

```
# ------------ Class for contrastive loss
class SupervisedContrastiveLoss(tf.keras.losses.Loss):
    def __init__(self, temperature= 0.05, name=None):
        super(SupervisedContrastiveLoss, self).__init__(name=name)
        self.temperature = temperature

    def __call__(self, labels, feature_vectors, sample_weight=None):
        # Normalize feature vectors
        feature_vectors_normalized = tf.math.l2_normalize(feature_vectors, axis=1)
        # Compute logits
        logits = tf.divide(
            tf.matmul(
                feature_vectors_normalized, tf.transpose(feature_vectors_normalized)
            ),
            self.temperature,
        )
        return npairs_loss(tf.squeeze(labels), logits)

class SCCAE(classifier):

    def __init__(self, param, name = "SCCAE"):
        super().__init__(param, name)

    def get_augmenter(self):

        return tf.keras.Sequential(
            [
                tf.keras.Input(shape= self.input_shape),
```

```
                # - window_warp
                tf.keras.layers.Lambda(lambda t: tf.numpy_function(window_warp, [t[:, :, :, 0], K.random_uniform((1,), 0.1, 0.3)[0]], [tf.float32])),
                tf.keras.layers.Lambda(lambda t: tf.expand_dims(t, -1)),

                # - window_slice
                tf.keras.layers.Lambda(lambda t: tf.numpy_function(window_slice, [t[:, :, :, 0], K.random_uniform((1,), 0.6, 0.9)[0]], [tf.float32])),
                tf.keras.layers.Lambda(lambda t: tf.expand_dims(t, -1)),

                # - crop
                #tf.keras.layers.Lambda(lambda t: tf.numpy_function(crop, [t[:, :, :, 0]], [tf.float32])),
                #tf.keras.layers.Lambda(lambda t: tf.expand_dims(t, -1)),
            ]
        )

    def visualize_augmentations(self, data):
        # Random sample from the dataset
        sample = data[random.randrange(len(data))]
        sample = data[0]

        augmenter = self.get_augmenter()

        augmented_data = augmenter(sample[np.newaxis, ...]).numpy()
        augmented_data2 = augmenter(sample[np.newaxis, ...]).numpy()

        #num_sensors = data.shape[2]

        for i in range(self.num_sensors):
```

```
        plt.figure(figsize=(16, 12), dpi=80)
        plt.plot(sample[:, i], label= "Original")
        plt.plot( augmented_data[0, :, i, 0], label= "Augmented 1")
        plt.plot(augmented_data2[0, :, i, 0], label= "Augmented 2")
        plt.legend()
        plt.tight_layout()
        plt.show()

def add_projection_head(self, encoder):
    inputs = tf.keras.Input(shape=self.input_shape)
    features = encoder(inputs)
    flat = layers.Flatten()(features)
    d = layers.Dense(100)(flat)
    d = layers.Dense(50)(d)
    outputs = layers.Dense(25)(d)
    model = Model(inputs=inputs, outputs=outputs, name="cifar-encoder_with_projection-
head")
    return model

def create_encoder(self, x_train, y_train, override= False):

    # return encoder when it already exists
    if (not override) and (Path(self.encoder_filename, "saved_model.pb").exists()):
        return tf.keras.models.load_model(self.encoder_filename, compile= False)

    # convert labels from one-hot encoding
    if y_train.ndim == 2:
        y_train = np.argmax(y_train, axis= 1)
```

```
self.input_shape = x_train.shape[1:]

# Create the save directory if needed
self.encoder_filename.mkdir(parents=True, exist_ok=True)

# Visualize the augmentation process
'''
self.visualize_augmentations(x_train)
'''

# ---------- Contrastive pretraining

# downscale block of the encoder
def down_scale_block(inputs, filters, kernel, pool):
    conv = layers.Conv2D(filters= filters, kernel_size= kernel, activation = 'relu', 
padding = 'same')(inputs)
    result = layers.MaxPooling2D(pool_size= pool)(conv)
    return result

# create encoder
inputs = tf.keras.Input(shape= self.input_shape)
down = down_scale_block(inputs,  filters= 64, kernel= ( 7, 1), pool= (4, 1))
down = down_scale_block(down,    filters= 32, kernel= (11, 1), pool= (4, 1))
encoded = down_scale_block(down, filters= 16, kernel= (11, 1), pool= (4, 1))
encoded = layers.Flatten()(encoded)
encoder =  Model(inputs, encoded, name= "encoder")

# create pretraining model
augmenter = self.get_augmenter()
```

```
        encoder_with_projection = self.add_projection_head(encoder)
        pretraining_model = tf.keras.Sequential(
            [
                tf.keras.Input(shape= self.input_shape),
                augmenter,
                encoder_with_projection
            ],
            name="pretraining_model",
        )

        # ensure there is a 'temp' param
        if "temp" not in self.param:
                self.param["temp"]= 0.05

        pretraining_model.compile(
            optimizer=tf.keras.optimizers.Adam(0.0001),
            loss=SupervisedContrastiveLoss(temperature= self.param["temp"]),
        )

        pretraining_model.fit(x=x_train, y=y_train, verbose= verbose, epochs=
self.param["epochs"], batch_size= self.param["bs"],)

        # Save the encoder
        encoder.save(self.encoder_filename)
        return encoder

    def create_model(self):
```

```
            self.encoder_filename = Path("models", self.param["dataset"], self.name,
str(self.sub_test[0]))

            encoder = self.create_encoder(self.x_train, self.y_train)

            # make sure there is a freeze param
            if "freeze" not in self.param:
                self.param["freeze"]= True

            # freeze the model if needed
            if self.param["freeze"]:
                for layer in encoder.layers:
                    layer.trainable= False

            # Supervised finetuning of the pretrained encoder
            self.model = tf.keras.Sequential(
                [
                    layers.Input(shape= self.x_train.shape[1:]),
                    encoder,
                    layers.Dense(100, activation= "relu"),
                    layers.Dense(self.num_classes, activation='softmax')
                ],
                name="pain_model",
            )

#----------------------------------------------------------------------------------------------------
# Gaf
#----------------------------------------------------------------------------------------------------
class gaf_mdk(classifier):
```

```
def __init__(self, param, name = "gaf_mdk"):
    super().__init__(param, name)

def data_processing(self):

    self.num_classes = len(np.unique(self.y_train))

    # -------------------------------------------
    # Create GAF

    from pyts.image import GramianAngularField
    self.param["GAF_size"] = 64
    image_size = self.param["GAF_size"]
    gasf = GramianAngularField(image_size=image_size, method='summation')
    gadf = GramianAngularField(image_size=image_size, method='difference')

    def extract_gaf(x):
        x_gasf = gasf.fit_transform(x)
        x_gadf = gadf.fit_transform(x)
        return np.stack([x_gasf, x_gadf], axis= -1)

    x_train_gaf = []
    x_test_gaf = []
    num_sensors = self.x_train.shape[2]
    for sensor in range(num_sensors):
        x_train_gaf.append(extract_gaf(self.x_train[:, :, sensor, 0]))
        x_test_gaf.append(extract_gaf(self.x_test[:, :, sensor, 0]))
```

```
        self.x_train_gaf = np.concatenate(x_train_gaf, axis= -1)
        self.x_test_gaf = np.concatenate(x_test_gaf, axis= -1)

        show_plot = False
        if show_plot:
            for i in range(self.x_train.shape[0]):
                fig, axs = plt.subplots(2)
                axs[0].imshow(x_train_gaf[i, :, :, 0], aspect="auto")
                axs[1].plot(self.x_train[i, :, 0, 0])
                plt.show()
        # -------------------------------------------

        assert not np.any(np.isnan(self.x_train))
        assert not np.any(np.isnan(self.x_test))

        self.x_train = self.x_train_gaf
        self.x_test = self.x_test_gaf

    def create_model(self):

        def mdk_module(input, filters= 1):

            # residual connection
            branches = [input]
            # 1 x 1 conv
            branches.append(layers.Convolution2D(filters= filters, kernel_size= (1, 1),
padding="same", strides= (1, 1))(input))

            # the different dilated blocks
```

```
        for dilation in [1, 2, 4]:
            x = layers.Convolution2D(filters= filters, kernel_size= (3, 3), padding="same", strides= (1, 1), dilation_rate= (dilation, dilation))(input)
            branches.append(layers.Convolution2D(filters= filters, kernel_size= (3, 3), padding="same", strides= (1, 1), dilation_rate= (dilation, dilation))(x))

        # add everything
        return layers.Add()(branches)

    inputs = layers.Input(self.x_train.shape[1:])

    if "#blocks" not in self.param:
        self.param["#blocks"]= 4

    x = inputs
    for _ in range(self.param["#blocks"]):
        x = mdk_module(x)

    x = layers.Flatten()(x)
    x = layers.Dense(250, activation= "relu")(x)
    x = layers.Dense(100, activation= "relu", name= "mdk_features")(x)
    out = layers.Dense(self.num_classes, activation='softmax')(x)

    self.model = Model(inputs = inputs, outputs = out, name= "main_model")

    return self.model

#-----------------------------------------------------------------------------------------------------
# transformer
```

```
# https://github.com/imics-lab/recurrence-with-self-attention
#----------------------------------------------------------------------------------------------------

class transformer(classifier):

	def __init__(self, param, name = "transformer"):
		super().__init__(param, name)

	def data_processing(self):
		if "split" not in self.param:
			self.param["split"]= 8

		self.x_train = np.concatenate(np.split(self.x_train, self.param["split"], axis= 1), axis= -2)
		self.x_test  = np.concatenate(np.split(self.x_test,  self.param["split"], axis= 1), axis= -2)

		self.x_train = np.swapaxes(self.x_train, 1, 2)[:, :, :, 0]
		self.x_test  = np.swapaxes(self.x_test , 1, 2)[:, :, :, 0]

	def create_model(self):
		def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):
			# Normalization and Attention
			x = layers.LayerNormalization(epsilon=1e-6)(inputs)
			x = layers.MultiHeadAttention(
				key_dim=head_size, num_heads=num_heads, dropout=dropout
			)(x, x)
			x = layers.Dropout(dropout)(x)
			res = x + inputs

			# Feed Forward Part
```

```python
        x = layers.LayerNormalization(epsilon=1e-6)(res)
        x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)
        x = layers.Dropout(dropout)(x)
        x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)
        return x + res

    def build_model(
        input_shape,
        head_size=128,
        num_heads=2,
        ff_dim=32,
        num_transformer_blocks=2,
        mlp_units=[100],
        mlp_dropout=0.4,
        dropout=0.2,
    ):
        inputs = layers.Input(shape=input_shape)
        x = inputs

        for _ in range(num_transformer_blocks):
            x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)

        x = layers.GlobalAveragePooling1D(data_format="channels_first")(x)
        x = layers.Flatten()(x)
        for dim in mlp_units:
            x = layers.Dense(dim, activation="relu")(x)
            x = layers.Dropout(mlp_dropout)(x)
        outputs = layers.Dense(self.num_classes, activation="softmax")(x)
        model =  Model(inputs, outputs)
```

```
            return model

        if "num_transformer_blocks" not in self.param:
            self.param["num_transformer_blocks"]= 3

        input_shape = self.x_train.shape[1:]
        self.model = build_model(input_shape= input_shape, num_transformer_blocks=
self.param["num_transformer_blocks"])
```