PyTSK

Yuqi Cui

May 01, 2022

TABLE OF CONTENTS

PyTSK is a package for conveniently developing a TSK-type fuzzy neural networks. It's dependencies are as follows:

- Scikit-learn [Necessary] for machine learning operations.
- Numpy [Necessary] for matrix computing operations.
- Scipy [Necessary] for matrix computing operations.
- PyTorch [Necessary] for constructing and training fuzzy neural networks.
- Faiss [Optional] a faster version for k-means clustering.

To run the code in quick start, you also need to install the following packages:

- PMLB for downloading datasets.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 Dependency

PyTSK is a package for conveniently developing a TSK-type fuzzy neural networks. It's dependencies are as follows:

- Scikit-learn [Necessary] for machine learning operations.
- Numpy [Necessary] for matrix computing operations.
- PyTorch [Necessary] for constructing and training fuzzy neural networks.
- Faiss [Optional] a faster version for k-means clustering.

To run the code in quick start, you also need to install the following packages:

- PMLB for downloading datasets.

1.2 Pip install

Install by pip:

```
pip install scikit-pytsk
```

Source code can be found in Github: https://github.com/YuqiCui/PyTSK

CHAPTER

TWO

QUICK START

2.1 Training with gradient descent

Complete code can be found at: https://github.com/YuqiCui/PyTSK/blob/master/quickstart_gradient_descent.py

Import everything you need:

```
import numpy as np
import torch.nn as nn
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from torch.optim import AdamW

from pytsk.gradient_descent.antecedent import AntecedentGMF, antecedent_init_center
from pytsk.gradient_descent.callbacks import EarlyStoppingACC
from pytsk.gradient_descent.training import Wrapper
from pytsk.gradient_descent.tsk import TSK
```

Prepare dataset:

```
# Prepare dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2)  # X: [n_samples,␣
↪n_features], y: [n_samples, 1]
n_class = len(np.unique(y))  # Num. of class

# split train-test
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print("Train on {} samples, test on {} samples, num. of features is {}, num. of class is
↪{}".format(
    x_train.shape[0], x_test.shape[0], x_train.shape[1], n_class
))

# Z-score
ss = StandardScaler()
x_train = ss.fit_transform(x_train)
x_test = ss.transform(x_test)
```

Define TSK parameters:

```
# Define TSK model parameters
n_rule = 30  # Num. of rules
lr = 0.01  # learning rate
weight_decay = 1e-8
consbn = False
order = 1
```

Construct TSK model, for example, HTSK model with LN-ReLU:

```
# --------- Define antecedent ------------
init_center = antecedent_init_center(x_train, y_train, n_rule=n_rule)
gmf = nn.Sequential(
    AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, high_dim=True, init_center=init_
↪center),
    nn.LayerNorm(n_rule),
    nn.ReLU()
)
# set high_dim=True is highly recommended.

# --------- Define full TSK model ------------
model = TSK(in_dim=X.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf,␣
↪order=order, precons=None)
```

Define optimizer, split train-val, define earlystopping callback:

```
# ----------------- optimizer ----------------------------
ante_param, other_param = [], []
for n, p in model.named_parameters():
    if "center" in n or "sigma" in n:
        ante_param.append(p)
    else:
        other_param.append(p)
optimizer = AdamW(
    [{'params': ante_param, "weight_decay": 0},
    {'params': other_param, "weight_decay": weight_decay},],
    lr=lr
)
# ----------------- split 10% data for earlystopping -----------------
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1)
# ----------------- define the earlystopping callback -----------------
EACC = EarlyStoppingACC(x_val, y_val, verbose=1, patience=20, save_path="tmp.pkl")
```

Train TSK model:

```
wrapper = Wrapper(model, optimizer=optimizer, criterion=nn.CrossEntropyLoss(),
              epochs=300, callbacks=[EACC])
wrapper.fit(x_train, y_train)
wrapper.load("tmp.pkl")
```

Evaluate model's performance:

```
y_pred = wrapper.predict(x_test).argmax(axis=1)
print("[TSK] ACC: {:.4f}".format(accuracy_score(y_test, y_pred)))
```

2.2 Training with fuzzy clustering

Complete code can be found at: https://github.com/YuqiCui/PyTSK/blob/master/quickstart_fuzzy_clustering.py

Import everything you need:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.linear_model import RidgeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

from pytsk.cluster import FuzzyCMeans
```

Prepare dataset:

```
# Prepare dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2)  # X: [n_samples,␣
↪n_features], y: [n_samples, 1]
n_class = len(np.unique(y))  # Num. of class

# split train-test
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print("Train on {} samples, test on {} samples, num. of features is {}, num. of class is
↪{}".format(
    x_train.shape[0], x_test.shape[0], x_train.shape[1], n_class
))

# Z-score
ss = StandardScaler()
x_train = ss.fit_transform(x_train)
x_test = ss.transform(x_test)
```

Define & train the TSK model:

```
# --------------- Fit and predict ---------------
n_rule = 20
model = Pipeline(
    steps=[
        ("GaussianAntecedent", FuzzyCMeans(n_rule, sigma_scale="auto", fuzzy_index="auto
↪")),
        ("Consequent", RidgeClassifier())
    ]
)

model.fit(x_train, y_train)
y_pred = model.predict(x_test)
print("ACC: {:.4f}".format(accuracy_score(y_test, y_pred)))
```

If you need analysis the input of consequent part:

```
# ---------------- get the input of consequent part for further analysis-----------------
antecedent = model.named_steps['GaussianAntecedent']
consequent_input = antecedent.transform(x_test)
```

If you need grid search all important parameters:

```
param_grid = {
    "Consequent__alpha": [0.01, 0.1, 1, 10, 100],
    "GaussianAntecedent__n_rule": [10, 20, 30, 40],
    "GaussianAntecedent__sigma_scale": [0.01, 0.1, 1, 10, 100],
    "GaussianAntecedent__fuzzy_index": ["auto", 1.8, 2, 2.2],
}
search = GridSearchCV(model, param_grid, n_jobs=2, cv=5, verbose=10)
search.fit(x_train, y_train)
y_pred = search.predict(x_test)
print("ACC: {:.4f}".format(accuracy_score(y_test, y_pred)))
```

Evaluate model's performance:

```
y_pred = wrapper.predict(x_test).argmax(axis=1)
print("[TSK] ACC: {:.4f}".format(accuracy_score(y_test, y_pred)))
```

Complete code can be found at: https://github.com/YuqiCui/PyTSK/quick_start.py

CHAPTER

THREE

MODELS & TECHNIQUE

3.1 TSK

A basic TSK fuzzy system is a combination of R rules, the r-th rule can be represented as:

Rule$_r$: IF x_1 is $X_{r,1}$ and ... and x_D is $X_{r,D}$
THEN $y = w_1x_1 + ... + w_Dx_D + b$,

where x_d is d-th input feature, $X_{r,d}$ is the membership function of the d-th input feature in the r-th rule. The IF part is called antecedent, the THEN part is called consequent in this package. The antecedent output the firing levels of the rules (for those who are not familiar with fuzzy systems, you can understand the firing levels as the attention weight of a Transformer/Mixture-of-experts(MoE) model), and the consequent part output the final prediction.

To define a TSK model, we need to define both antecedent and consequent modules:

```
# --------- Data format ------------
# X: feature matrix, [n_data, n_dim] each row represents a sample with n_dim features
# y: label matrix, [n_data, 1]

# --------- Define TSK model parameters ------------
n_rule = 10 # define num. of rules
n_class = 2  # define num. of class (model output dimension)
order = 1  # 0 or 1, zero-order TSK model or first-order TSK model

# --------- Define antecedent ------------
# run kmeans clustering to get initial rule centers
init_center = antecedent_init_center(X, y, n_rule=n_rule)
# define the antecedent Module
gmf = AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, init_center=init_center)

# --------- Define full TSK model ------------
model = TSK(in_dim=X.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf,␣
↪order=order)
```

3.2 HTSK

Traditional TSK model tends to fail on high-dimensional problems, so **the HTSK (high-dimensional TSK) model is recommended for handling any-dimension problems**. More details about the HTSK model can be found in [1].

To define a HTSK model, we need to set `high_dim=True` when define antecedent:

```
init_center = antecedent_init_center(X, y, n_rule=n_rule)
gmf = AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, init_center=init_center, high_
↪dim=True)
```

[1] Cui Y, Wu D, Xu Y. Curse of dimensionality for tsk fuzzy neural networks: Explanation and solutions[C]//2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021: 1-8.

3.3 DropRule

Similar as Dropout, randomly dropping rules of TSK (DropRule) can improve the performance of TSK models [2,3,4].

To use DropRule, we need to add a Dropout layer after the antecedent output:

```
# --------- Define antecedent ------------
init_center = antecedent_init_center(X, y, n_rule=n_rule)
gmf = nn.Sequential(
    AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, high_dim=True, init_center=init_
↪center),
    nn.Dropout(p=0.25)
)

# --------- Define full TSK model ------------
model = TSK(in_dim=X.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf,␣
↪order=order)
```

[2] Wu D, Yuan Y, Huang J, et al. Optimize TSK fuzzy systems for regression problems: Minibatch gradient descent with regularization, DropRule, and AdaBound (MBGD-RDA)[J]. IEEE Transactions on Fuzzy Systems, 2019, 28(5): 1003-1015.

[3] Shi Z, Wu D, Guo C, et al. FCM-RDpA: tsk fuzzy regression model construction using fuzzy c-means clustering, regularization, droprule, and powerball adabelief[J]. Information Sciences, 2021, 574: 490-504.

[4] Guo F, Liu J, Li M, et al. A Concise TSK Fuzzy Ensemble Classifier Integrating Dropout and Bagging for High-dimensional Problems[J]. IEEE Transactions on Fuzzy Systems, 2021.

3.4 Batch Normalization

Batch normalization (BN) can be used to normalize the input of consequent parameters, and the experiments in [5] have shown that BN can speed up the convergence and improve the performance of a TSK model.

To add the BN layer, we need to set `precons=nn.BatchNorm1d(in_dim)` when defining the TSK model:

```
model = TSK(in_dim=X.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf,␣
↪order=order, precons=nn.BatchNorm1d(in_dim))
```

[5] Cui Y, Wu D, Huang J. Optimize tsk fuzzy systems for classification problems: Minibatch gradient descent with uniform regularization and batch normalization[J]. IEEE Transactions on Fuzzy Systems, 2020, 28(12): 3065-3075.

3.5 Uniform Regularization

[5] also proposed a uniform regularization, which can mitigate the "winner gets all" problem when training TSK with mini-batch gradient descent algorithms. The "winner gets all" problem will cause only a small number of rules dominant the prediction, other rules will have nearly zero contribution to the prediction. The uniform regularization loss is:

$$\ell_{UR} = \sum_{r=1}^{R}(\frac{1}{N}\sum_{n=1}^{N} f_{n,r} - \tau)^2,$$

where $f_{n,r}$ represents the firing level of the n-th sample on the r-th rule, N is the batch size. Experiments in [5] has proved that UR can significantly improve the performance of TSK fuzzy system. The UR loss is defined at `ur_loss`. If you want to use the UR during training, you can simply set a positive UR weight when initialize `Wrapper`:

```
ur = 1.  # must > 0
ur_tau  # a float number between 0 and 1
wrapper = Wrapper(
    model, optimizer=optimizer, criterion=criterion, epochs=epochs, callbacks=callbacks,␣
↪ur=ur, ur_tau=ur_tau
)
```

[5] Cui Y, Wu D, Huang J. Optimize tsk fuzzy systems for classification problems: Minibatch gradient descent with uniform regularization and batch normalization[J]. IEEE Transactions on Fuzzy Systems, 2020, 28(12): 3065-3075.

3.6 Layer Normalization

Layer normalization (LN) can be used to normalize the firing levels of the antecedent part [6]. It can be easily proved that the scale of firing levels will decrease when more rules are used. Since the gradient of the parameters in TSK are all relevant with the firing level, it will cause a gradient vanishing problem, making TSKs perform bad, especially when using SGD/SGDM as optimizer. LN normalizes the firing level, similar as the LN layer in Transformer, can solve the gradient vanishing problems and improve the performance. Adding a ReLU acitivation can further filter the negative firing levels generated by LN, improving the interpretability and robustness to outliers.

To add LN & ReLU, we can do:

```
# --------- Define antecedent ------------
init_center = antecedent_init_center(X, y, n_rule=n_rule)
gmf = nn.Sequential(
    AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, high_dim=True, init_center=init_
↪center),
    nn.LayerNorm(n_rule),
    nn.ReLU()
)

# --------- Define full TSK model ------------
model = TSK(in_dim=x_train.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf,␣
↪order=order)
```

[6] *Cui Y, Wu D, Xu Y, Peng R. Layer Normalization for TSK Fuzzy System Optimization in Regression Problems[J]. IEEE Transactions on Fuzzy Systems, submitted.*

3.7 Deep learning

TSK models can also be used as a classifier/regressor in a deep neural network, which may improving the performance of neural networks. To do that, we first need to get the middle output of neural networks for antecedent initialization, and then define the deep fuzzy systems as follows:

```
# --------- Define antecedent ------------
# Note that X should be the output of NeuralNetworks, y is still the corresponding label
init_center = antecedent_init_center(X, y, n_rule=n_rule)
gmf = AntecedentGMF(in_dim=X.shape[1], n_rule=n_rule, high_dim=True, init_center=init_
↪center)

# ---------- Define deep learning + TSK -------------
model = nn.Sequential(
    NeuralNetworks(),
    TSK(in_dim=X.shape[1], out_dim=n_class, n_rule=n_rule, antecedent=gmf, order=order),
)
```

CHAPTER

FOUR

API: PYTSK.CLUSTER

`pytsk.cluster` mainly provide fuzzy clustering algorithms. Each algorithm will implemente a `transform` method, which convert the input of raw feature $X \in \mathbb{R}^{N,D}$ into the consequent input matrix $P \in \mathbb{R}^{N,T}$ of TSK fuzzy systems, where D is the input dimension, N is the number of samples, for a zero order TSK fuzzy system, $T = R$, where R is the number of rules (equal to the number of clusters of the fuzzy clustering algorithm), for a first-order TSK fuzzy system, $T = (D + 1) \times R$.

class `pytsk.cluster.`**`BaseFuzzyClustering`**

Parent: `object`.

The parent class of fuzzy clustering classes.

`set_params`(*self*, ***params*)

Setting attributes. Implemented to adapt the API of scikit-learn.

class `pytsk.cluster.`**`FuzzyCMeans`**(*n_cluster*, *fuzzy_index='auto'*, *sigma_scale='auto'*, *init='random'*, *tol_iter=100*, *error=1e-06*, *dist='euclidean'*, *verbose=0*, *order=1*)

Parent: *`BaseFuzzyClustering`*, `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`.

The fuzzy c-means (FCM) clustering algorithm [1]. This implementation is adopted from the scikit-fuzzy package. When constructing a TSK fuzzy system, a fuzzy clustering algorithm is usually used to compute the antecedent parameters, after that, the consequent parameters can be computed by least-squared error algorithms, such as Ridge regression [2]. How to use this class can be found at Quick start.

The objective function of the FCM is:

$$J = \sum_{i=1}^{N} \sum_{j=1}^{C} U_{i,j}^{m} \|\mathbf{x}_i - \mathbf{v}_j\|_2^2$$

$$s.t. \sum_{j=1}^{C} \mu_{i,j} = 1, i = 1, ..., N,$$

where N is the number of samples, C is the number of clusters (which also corresponding to the number of rules of TSK fuzzy systems), m is the fuzzy index, $\mathbf{x}_i$ is the i-th input vector, $\mathbf{v}_j$ is the j-th cluster center vector, $U_{i,j}$ is the membership degree of the i-th input vector on the j-th cluster center vector. The FCM algorithm will obtain the centers $\mathbf{v}_j, j = 1, ..., C$ and the membership degrees $U_{i,j}$.

Parameters

- **`n_cluster`** (*`int`*) – Number of clusters, equal to the number of rules R of a TSK model.
- **`fuzzy_index`** (*`float/str`*) – Fuzzy index of the FCM algorithm, default *auto*. If `fuzzy_index=auto`, then the fuzzy index is computed as $\min(N, D-1)/(\min(N, D-1) - 2)$ (If $\min(N, D-1) < 3$, fuzzy index will be set to 2), according to [3]. Otherwise the given float value is used.

- **sigma_scale** (*float/str*) – The scale parameter h to adjust the actual standard deviation σ of the Gaussian membership function in TSK antecedent part. If `sigma_scale=auto`, `sigma_scale` will be set as $\sqrt{D}$, where D is the input dimension [4]. Otherwise the given float value is used.
- **init** (*str/np.array*) – The initialization strategy of the membership grid matrix U. Support "random" or numpy array with the size of $[R, N]$, where R is the number of clusters/rules, N is the number of training samples. If `init="random"`, the initial membership grid matrix will be randomly initialized, otherwise the given matrix will be used.
- **tol_iter** (*int*) – The total iteration of the FCM algorithm.
- **error** (*float*) – The maximum error that will stop the iteration before maximum iteration is reached.
- **dist** (*str*) – The distance type for the `scipy.spatial.distance.cdist()` function, default "euclidean". The distance function can also be "braycurtis", "canberra", "chebyshev", "cityblock", "correlation", "cosine", "dice", "euclidean", "hamming", "jaccard", "jensenshannon", "kulsinski", "kulczynski1", "mahalanobis", "matching", "minkowski", "rogerstanimoto", "russellrao", "seuclidean", "sokalmichener", "sokalsneath", "sqeuclidean", "yule".
- **verbose** (*int*) – If > 0, it will show the loss of the FCM objective function during iterations.
- **order** (*int*) – 0 or 1. Decide whether to construct a zero-order TSK or a first-order TSK.

fit(*self*, *X*, *y=None*)

Run the FCM algorithm.

Parameters

- **X** (*numpy.array*) – Input array with the size of $[N, D]$, where N is the number of training samples, and D is number of features.
- **y** (*numpy.array*) – Not used. Pass None.

predict(*self*, *X*, *y=None*)

Predict the membership degrees of **X** on each cluster.

Parameters

- **X** (*numpy.array*) – Input array with the size of $[N, D]$, where N is the number of training samples, and D is number of features.
- **y** (*numpy.array*) – Not used. Pass None.

Returns return the membership degree matrix U with the size of $[N, R]$, where N is the number of samples of **X**, and R is the number of clusters/rules. $U_{i,j}$ represents the membership degree of the i-th sample on the r-th cluster.

transform(*self*, *X*, *y=None*)

Compute the membership degree matrix U, and use X and U to get the consequent input matrix P using function `x2xp(x, u, order)`

Parameters

- **X** (*numpy.array*) – Input array with the size of $[N, D]$, where N is the number of training samples, and D is number of features.
- **y** (*numpy.array*) – Not used. Pass None.

Returns return the consequent input X_p with the size of $[N, (D+1) \times R]$, where N is the number of test samples, D is number of features, R is the number of clusters/rules.

x2xp(*X*, *U*, *order*)

Convert the feature matrix X and the membership degree matrix U into the consequent input matrix X_p

Each row in $X \in \mathbb{R}^{N,D}$ represents a D-dimension input vector. Suppose vector $\mathbf{x}$ is one row, and then the consequent input matrix P is computed as [5] for a first-order TSK:

$$\begin{aligned}
\mathbf{x}_e &= (1, \mathbf{x}), \\
\tilde{\mathbf{x}}_r &= u_r \mathbf{x}_e, \\
\mathbf{p} &= (\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, ..., \tilde{\mathbf{x}}_R),
\end{aligned}$$

where $\mathbf{p}$ is the corresponding row in P, which is a $(D+1)\times R$-dimension vector. Then the consequent parameters of TSK can be optimized by any linear regression algorithms.

Parameters

- **x** (*numpy.array*) – size: $[N, D]$. Input features.
- **u** (*numpy.array*) – size: $[N, R]$. Corresponding membership degree matrix.
- **order** (*int*) – 0 or 1. The order of TSK models.

Returns If `order=0`, return U directly, else if `order=1`, return the matrix X_p with the size of $[N, (D+1) \times R]$. Details can be found at [2].

compute_variance(*X*, *U*, *V*)

Compute the variance of the Gaussian membership function in TSK fuzzy systems. After performing the FCM, one can use $\mathbf{v}_j$ and $U_{i,j}$ to construct the Gaussian membership function based antecedent of a TSK fuzzy system. The center of the Gaussian membership function can be directly set as center *mathbf{v}_j*, the standard deviation of the Gaussian membership function can be computed as follows:

$$\sigma_{r,d} = \left[\sum_{i=1}^{N} U_{i,r} (x_{i,d} - v_{r,d})^2 / \sum_{i=1}^{N} U_{i,r} \right]^{1/2},$$

where $v_{r,d}$ represents the cluster center of the d-th dimension in the r-th rule.

Parameters

- **x** (*numpy.array*) – Input matrix X with the size of $[N, D]$.
- **u** (*numpy.array*) – Membership degree matrix U with the size of $[R, N]$.
- **v** (*numpy.array*) – Cluster center matrix V with the size of $[R, D]$.

Returns The standard variation matrix Σ with the size of $[R, D]$.

[1] Bezdek J C, Ehrlich R, Full W. FCM: The fuzzy c-means clustering algorithm[J]. Computers & geosciences, 1984, 10(2-3): 191-203.

[2] Wang S, Chung K F L, Zhaohong D, et al. Robust fuzzy clustering neural network based on -insensitive loss function[J]. Applied Soft Computing, 2007, 7(2): 577-584.

[3] Yu J, Cheng Q, Huang H. Analysis of the weighting exponent in the FCM[J]. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 2004, 34(1): 634-639.

[4] Cui Y, Wu D, Xu Y. Curse of dimensionality for tsk fuzzy neural networks: Explanation and solutions[C]//2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021: 1-8.

[5] Deng Z, Choi K S, Chung F L, et al. Scalable TSK fuzzy modeling for very large datasets using minimal-enclosing-ball approximation[J]. IEEE Transactions on Fuzzy Systems, 2010, 19(2): 210-226.

CHAPTER

FIVE

API: PYTSK.GRADIENT_DESCENT

This package contains all the APIs you need to build and train a fuzzy neural networks.

5.1 pytsk.gradient_descent.antecedent

class AntecedentGMF(*in_dim*, *n_rule*, *high_dim=False*, *init_center=None*, *init_sigma=1.0*, *eps=1e-08*)

Parent: `torch.nn.Module`

The antecedent part with Gaussian membership function. Input: data, output the corresponding firing levels of each rule. The firing level $f_r(\mathbf{x})$ of the r-th rule are computed by:

$$\mu_{r,d}(x_d) = \exp(-\frac{(x_d - m_{r,d})^2}{2\sigma_{r,d}^2}),$$

$$f_r(\mathbf{x}) = \prod_{d=1}^{D} \mu_{r,d}(x_d),$$

$$\overline{f}_r(\mathbf{x}) = \frac{f_r(\mathbf{x})}{\sum_{i=1}^{R} f_i(\mathbf{x})}.$$

Parameters

- **in_dim** (*int*) – Number of features D of the input.
- **n_rule** (*int*) – Number of rules R of the TSK model.
- **high_dim** (*bool*) – Whether to use the HTSK defuzzification. If `high_dim=True`, HTSK is used. Otherwise the original defuzzification is used. More details can be found at [1]. TSK model tends to fail on high-dimensional problems, so set `high_dim=True` is highly recommended for any-dimensional problems.
- **init_center** (*numpy.array*) – Initial center of the Gaussian membership function with the size of $[D, R]$. A common way is to run a KMeans clustering and set `init_center` as the obtained centers. You can simply run *pytsk.gradient_descent.antecedent.antecedent_init_center* to obtain the center.
- **init_sigma** (*float*) – Initial σ of the Gaussian membership function.
- **eps** (*float*) – A constant to avoid the division zero error.

init(*self*, *center*, *sigma*)

Change the value of `init_center` and `init_sigma`.

Parameters

- **center** (*numpy.array*) – Initial center of the Gaussian membership function with the size of $[D, R]$. A common way is to run a KMeans clustering and set `init_center` as the obtained centers. You can simply run *pytsk.gradient_descent.antecedent.antecedent_init_center* to obtain the center.
- **sigma** (*float*) – Initial σ of the Gaussian membership function.

reset_parameters(*self*)

Re-initialize all parameters.

forward(*self*, *X*)

Forward method of Pytorch Module.

Parameters **X** (*torch.tensor*) – Pytorch tensor with the size of $[N, D]$, where N is the number of samples, D is the input dimension.

Returns Firing level matrix U with the size of $[N, R]$.

class AntecedentShareGMF(*in_dim*, *n_mf=2*, *high_dim=False*, *init_center=None*, *init_sigma=1.0*, *eps=1e-08*)

Parent: `torch.nn.Module`

The antecedent part with Gaussian membership function, rules will share the membership functions on each feature [2]. The number of rules will be M^D, where M is `n_mf`, D is the number of features (`in_dim`).

Parameters

- **in_dim** (*int*) – Number of features D of the input.
- **n_mf** (*int*) – Number of membership functions M of each feature.
- **high_dim** (*bool*) – Whether to use the HTSK defuzzification. If `high_dim=True`, HTSK is used. Otherwise the original defuzzification is used. More details can be found at [1]. TSK model tends to fail on high-dimensional problems, so set `high_dim=True` is highly recommended for any-dimensional problems.
- **init_center** (*numpy.array*) – Initial center of the Gaussian membership function with the size of $[D, M]$.
- **init_sigma** (*float*) – Initial σ of the Gaussian membership function.
- **eps** (*float*) – A constant to avoid the division zero error.

init(*self*, *center*, *sigma*)

Change the value of `init_center` and `init_sigma`.

Parameters

- **center** (*numpy.array*) – Initial center of the Gaussian membership function with the size of $[D, M]$.
- **sigma** (*float*) – Initial σ of the Gaussian membership function.

reset_parameters(*self*)

Re-initialize all parameters.

forward(*self*, *X*)

Forward method of Pytorch Module.

Parameters **X** (*torch.tensor*) – Pytorch tensor with the size of $[N, D]$, where N is the number of samples, D is the input dimension.

Returns Firing level matrix U with the size of $[N, R]$, $R = M^D$:.

class AntecedentTriMF(*in_dim*, *n_rule*, *init_center=None*, *init_left_dist=3.0*, *init_right_dist=3.0*, *eps=1e-08*)

Parent: `torch.nn.Module`

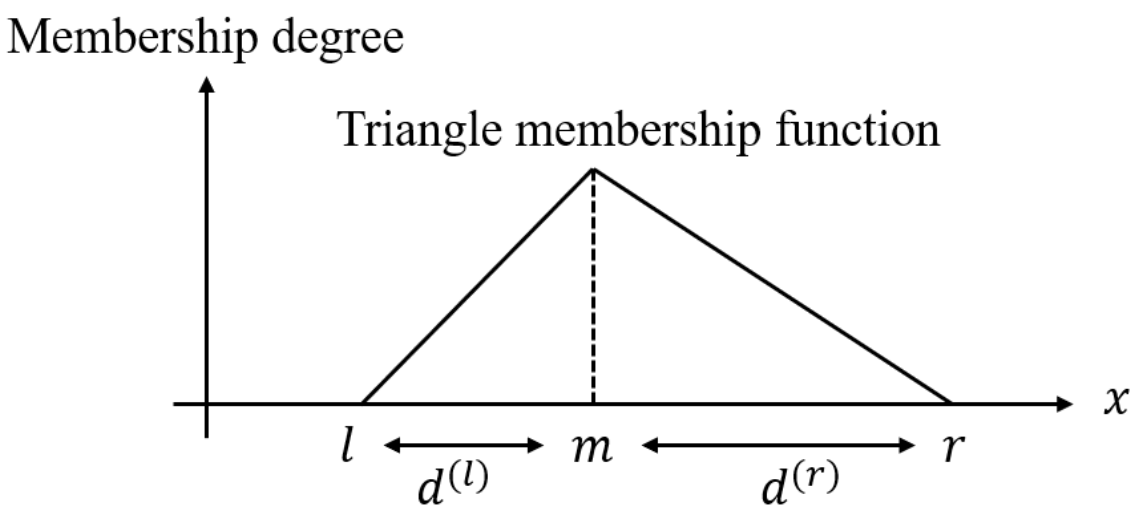
The antecedent part with triangle membership function. Input: data, output the corresponding firing levels of each rule. The firing level $f_r(\mathbf{x})$ of the r-th rule are computed by:

$$\mu_{r,d}(x_d) = \max(0, \min(\frac{1}{d_{r,d}^{(l)}}x + 1 - \frac{m_{r,d}}{d_{r,d}^{(l)}}, -\frac{1}{d_{r,d}^{(r)}}x + 1 + \frac{m_{r,d}}{d_{r,d}^{(r)}})),$$

$$f_r(\mathbf{x}) = \prod_{d=1}^{D} \mu_{r,d}(x_d),$$

$$\overline{f}_r(\mathbf{x}) = \frac{f_r(\mathbf{x})}{\sum_{i=1}^{R} f_i(\mathbf{x})},$$

where $d_{r,d}^{(l)}/d_{r,d}^{(r)}$ is the distance between the center and the left/right vertex of the triangle membership function, respectively. A simple schematic is shown below:



Parameters

- **in_dim** (*int*) – Number of features D of the input.
- **n_rule** (*int*) – Number of rules R of the TSK model.
- **init_center** (*numpy.array*) – Initial center of the triangle membership function with the size of $[D, M]$.
- **init_left_dist** (*float*) – Initial $d^{(l)}$ of the triangle membership function.
- **init_right_dist** (*float*) – Initial $d^{(r)}$ of the triangle membership function.
- **eps** (*float*) – A constant to avoid the division zero error.

init(*self*, *center*, *left_dist*, *right_dist*)

Change the value of `init_center`, `init_left_dist` and `init_right_dist`.

Parameters

- **init_center** (*numpy.array*) – Initial center of the triangle membership function with the size of $[D, R]$.
- **left_dist** (*float*) – Initial $d^{(l)}$ of the triangle membership function.

- **right_dist** (*float*) – Initial $d^{(r)}$ of the triangle membership function.

reset_parameters(*self*)

Re-initialize all parameters.

forward(*self*, *X*)

Forward method of Pytorch Module.

Parameters **X** (*torch.tensor*) – Pytorch tensor with the size of $[N, D]$, where N is the number of samples, D is the input dimension.

Returns Firing level matrix U with the size of $[N, R]$:.

class AntecedentShareTriMF(*in_dim*, *n_mf=2*, *init_center=None*, *init_left_dist=3.0*, *init_right_dist=3.0*, *eps=1e-08*)

Parent: `torch.nn.Module`

The antecedent part with triangle membership function, rules will share the membership functions on each feature [2]. The number of rules will be M^D, where M is `n_mf`, D is the number of features (`in_dim`).

Parameters

- **in_dim** (*int*) – Number of features D of the input.
- **n_mf** (*int*) – Number of membership functions M of each feature.
- **init_center** (*numpy.array*) – Initial center of the triangle membership function with the size of $[D, M]$.
- **init_left_dist** (*float*) – Initial $d^{(l)}$ of the triangle membership function.
- **init_right_dist** (*float*) – Initial $d^{(r)}$ of the triangle membership function.
- **eps** (*float*) – A constant to avoid the division zero error.

init(*self*, *center*, *sigma*)

Change the value of `init_center`, `init_left_dist` and `init_right_dist`.

Parameters

- **init_center** (*numpy.array*) – Initial center of the triangle membership function with the size of $[D, M]$.
- **left_dist** (*float*) – Initial $d^{(l)}$ of the triangle membership function.
- **right_dist** (*float*) – Initial $d^{(r)}$ of the triangle membership function.

reset_parameters(*self*)

Re-initialize all parameters.

forward(*self*, *X*)

Forward method of Pytorch Module.

Parameters **X** (*torch.tensor*) – Pytorch tensor with the size of $[N, D]$, where N is the number of samples, D is the input dimension.

Returns Firing level matrix U with the size of $[N, R]$, $R = M^D$:.

antecedent_init_center(*X*, *y=None*, *n_rule=2*, *method='kmean'*, *engine='sklearn'*, *n_init=20*)

This function run KMeans clustering to obtain the `init_center` for *AntecedentGMF()*.

```
>>> init_center = antecedent_init_center(X, n_rule=10, method="kmean", n_init=20)
>>> antecedent = AntecedentGMF(X.shape[1], n_rule=10, init_center=init_center)
```

Parameters

- **X** (*numpy.array*) – Feature matrix with the size of $[N, D]$, where N is the number of samples, D is the number of features.
- **y** (*numpy.array*) – None, not used.
- **n_rule** (*int*) – Number of rules R. This function will run a KMeans clustering to obtain R cluster centers as the initial antecedent center for TSK modeling.
- **method** (*str*) – Current version only support "kmean".
- **engine** (*str*) – "sklearn" or "faiss". If "sklearn", then the `sklearn.cluster.KMeans()` function will be used, otherwise the `faiss.Kmeans()` will be used. Faiss provide a faster KMeans clustering algorithm, "faiss" is recommended for large datasets.
- **n_init** (*int*) – Number of initialization of the KMeans algorithm. Same as the parameter `n_init` in `sklearn.cluster.KMeans()` and the parameter `nredo` in `faiss.Kmeans()`.

[1] Cui Y, Wu D, Xu Y. Curse of dimensionality for tsk fuzzy neural networks: Explanation and solutions[C]//2021 International Joint Conference on Neural Networks (IJCNN). IEEE, 2021: 1-8.

[2] Shi Y, Mizumoto M. A new approach of neuro-fuzzy learning algorithm for tuning fuzzy rules[J]. Fuzzy sets and systems, 2000, 112(1): 99-116.

5.2 pytsk.gradient_descent.tsk

class TSK(*in_dim*, *out_dim*, *n_rule*, *antecedent*, *order=1*, *eps=1e-08*, *precons=None*)

Parent: `torch.nn.Module`

This module define the consequent part of the TSK model and combines it with a pre-defined antecedent module. The input of this module is the raw feature matrix, and output the final prediction of a TSK model.

Parameters

- **in_dim** (*int*) – Number of features D.
- **out_dim** (*int*) – Number of output dimension C.
- **n_rule** (*int*) – Number of rules R, must equal to the `n_rule` of the `Antecedent()`.
- **antecedent** (*torch.Module*) – An antecedent module, whose output dimension should be equal to the number of rules R.
- **order** (*int*) – 0 or 1. The order of TSK. If 0, zero-oder TSK, else, first-order TSK.
- **eps** (*float*) – A constant to avoid the division zero error.
- **consbn** (*torch.nn.Module*) – If none, the raw feature will be used as the consequent input; If a pytorch module, then the consequent input will be the output of the given module. If you wish to use the BN technique we mentioned in Models & Technique, you can set `precons=nn.BatchNorm1d(in_dim)`.

reset_parameters(*self*)

Re-initialize all parameters, including both consequent and antecedent parts.

forward(*self*, *X*, *get_frs=False*)

Parameters

- **X** (*torch.tensor*) – Input matrix with the size of $[N, D]$, where N is the number of samples.
- **get_frs** (*bool*) – If true, the firing levels (the output of the antecedent) will also be returned.

Returns If `get_frs=True`, return the TSK output $Y \in \mathbb{R}^{N,C}$ and the antecedent output $U \in \mathbb{R}^{N,R}$. If `get_frs=False`, only return the TSK output Y.

5.3 pytsk.gradient_descent.training

ur_loss(*frs*, *tau=0.5*)

The uniform regularization (UR) proposed by Cui et al. [3]. UR loss is computed as $\ell_{UR} = \sum_{r=1}^{R}(\frac{1}{N}\sum_{n=1}^{N} f_{n,r} - \tau)^2$, where $f_{n,r}$ represents the firing level of the n-th sample on the r-th rule.

Parameters

- **frs** (*torch.tensor*) – The firing levels (output of the antecedent) with the size of $[N, R]$, where N is the number of samples, R is the number of ruels.
- **tau** (*float*) – The expectation τ of the average firing level for each rule. For a C-class classification problem, we recommend setting τ to $1/C$, for a regression problem, τ can be set as 0.5.

Returns A scale value, representing the UR loss.

class Wrapper(*model*, *optimizer*, *criterion*, *batch_size=512*, *epochs=1*, *callbacks=None*, *label_type='c'*, *device='cpu'*, *reset_param=True*, *ur=0*, *ur_tau=0.5*, ***kwargs*)

This class provide a training framework for beginners to train their fuzzy neural networks.

Parameters

- **model** (*torch.nn.Module*) – The pre-defined TSK model.
- **optimizer** (*torch.Optimizer*) – Pytorch optimizer.
- **torch.nn._Loss** – Pytorch loss. For example, `torch.nn.CrossEntropyLoss()` for classification tasks, and `torch.nn.MSELoss()` for regression tasks.
- **batch_size** (*int*) – Batch size during training & prediction.
- **epochs** (*int*) – Training epochs.
- **callbacks** (*[Callback]*) – List of callbacks.
- **label_type** (*str*) – Label type, "c" or "r", when `label_type="c"`, label's dtype will be changed to "int64", when `label_type="r"`, label's dtype will be changed to "float32".

```
>>> from pytsk.gradient_descent import antecedent_init_center, AntecedentGMF, TSK,␣
↪EarlyStoppingACC, EvaluateAcc, Wrapper
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import StandardScaler
>>> from torch.optim import AdamW
```

(continues on next page)

(continued from previous page)

```
>>> import torch.nn as nn
>>> # ----------------- define data -----------------
>>> X, y = make_classification(random_state=0)
>>> x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
>>> ss = StandardScaler()
>>> x_train = ss.fit_transform(x_train)
>>> x_test = ss.transform(x_test)
>>> # ----------------- define TSK model -----------------
>>> n_rule = 10  # define number of rules
>>> n_class = 2  # define output dimension
>>> order = 1  # first-order TSK is used
>>> consbn = True  # consbn tech is used
>>> weight_decay = 1e-8  # weight decay for pytorch optimizer
>>> lr = 0.01  # learning rate for pytorch optimizer
>>> init_center = antecedent_init_center(x_train, y_train, n_rule=n_rule)  # obtain␣
↪the initial antecedent center
>>> gmf = AntecedentGMF(in_dim=x_train.shape[1], n_rule=n_rule, high_dim=True, init_
↪center=init_center)  # define antecedent
>>>  model = TSK(in_dim=x_train.shape[1], out_dim=n_class, n_rule=n_rule,␣
↪antecedent=gmf, order=order, consbn=consbn)  # define TSK
>>> # ----------------- define optimizers -----------------
>>> ante_param, other_param = [], []
>>> for n, p in model.named_parameters():
>>>     if "center" in n or "sigma" in n:
>>>         ante_param.append(p)
>>>     else:
>>>         other_param.append(p)
>>> optimizer = AdamW(
>>>     [{'params': ante_param, "weight_decay": 0},  # antecedent parameters␣
↪usually don't need weight_decay
>>>     {'params': other_param, "weight_decay": weight_decay},],
>>>     lr=lr
>>> )
>>> # ----------------- split 20% data for earlystopping -----------------
>>> x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.
↪2)
>>> # ----------------- define the earlystopping callback -----------------
>>> EACC = EarlyStoppingACC(x_val, y_val, verbose=1, patience=40, save_path="tmp.pkl
↪")  # Earlystopping
>>> TACC = EvaluateAcc(x_test, y_test, verbose=1)  # Check test acc during training
>>> # ----------------- train model -----------------
>>> wrapper = Wrapper(model, optimizer=optimizer, criterion=nn.CrossEntropyLoss(),
>>>                epochs=300, callbacks=[EACC, TACC], ur=0, ur_tau=1/n_class)  #␣
↪define training wrapper, ur weight is set to 0
>>> wrapper.fit(x_train, y_train)  # fit
>>> wrapper.load("tmp.pkl")  # load best model saved by EarlyStoppingACC callback
>>> y_pred = wrapper.predict(x_test).argmax(axis=1)  # predict, argmax for␣
↪extracting classfication label
>>> print("[TSK] ACC: {:.4f}".format(accuracy_score(y_test, y_pred)))  # print ACC
```

train_on_batch(*self*, *input*, *target*)

Define how to update a model with one batch of data. This method can be overwrite for custom training

strategy.

Parameters

- **input** (*torch.tensor*) – Feature matrix with the size of $[N, D]$, N is the number of samples, D is the input dimension.
- **target** (*torch.tensor*) – Target matrix with the size of $[N, C]$, C is the output dimension.

fit(*X*, *y*)

Train the `model` with numpy array.

Parameters

- **X** (*numpy.array*) – Feature matrix X with the size of $[N, D]$.
- **y** (*numpy.array*) – Label matrix Y with the size of $[N, C]$, for classification task, $C = 1$, for regression task, C is the number of the output dimension of `model`.

fit_loader(*self*, *train_loader*)

Train the `model` with user-defined pytorch dataloader.

Parameters **train_loader** (*torch.utils.data.DataLoader*) – Data loader, the output of the loader should be corresponding to the inputs of *train_on_batch*. For example, if dataloader has two output, then *train_on_batch* should also have two inputs.

predict(*self*, *X*, *y=None*)

Get the prediction of the model.

Parameters

- **X** (*numpy.array*) – Feature matrix X with the size of $[N, D]$.
- **y** – Not used.

Returns Prediction matrix $\hat{Y}$ with the size of $[N, C]$, C is the output dimension of the `model`.

predict_proba(*self*, *X*, *y=None*)

For classification problem only, need `label_type="c"`, return the prediction after softmax.

Parameters

- **X** (*numpy.array*) – Feature matrix X with the size of $[N, D]$.
- **y** – Not used.

Returns Prediction matrix $\hat{Y}$ with the size of $[N, C]$, C is the output dimension of the `model`.

save(*self*, *path*)

Save model.

Parameters **path** (*str*) – Model save path.

load(*self*, *path*)

Load model.

Parameters **path** (*str*) – Model save path.

[3] Cui Y, Wu D, Huang J. Optimize tsk fuzzy systems for classification problems: Minibatch gradient descent with uniform regularization and batch normalization[J]. IEEE Transactions on Fuzzy Systems, 2020, 28(12): 3065-3075.

5.4 pytsk.gradient_descent.callbacks

class Callback

Similar as the callback class in Keras, our package provides a simplified version of callback, which allow users to monitor metrics during the training. We strongly recommend uses to custom their callbacks, here we provide two examples, *EvaluateAcc* and *EarlyStoppingACC*.

on_batch_begin(*self*, *wrapper*)

Will be called before each batch.

on_batch_end(*self*, *wrapper*)

Will be called after each batch.

on_epoch_begin(*self*, *wrapper*)

Will be called before each epoch.

on_epoch_end(*self*, *wrapper*)

Will be called after each epoch.

class EvaluateAcc(*X*, *y*, *verbose=0*)

Evaluate the accuracy during training.

Parameters

- **X** (*numpy.array*) – Feature matrix with the size of $[N, D]$.
- **y** (*numpy.array*) – Label matrix with the size of $[N, 1]$.

on_epoch_end(*self*, *wrapper*)

```
>>> def on_epoch_end(self, wrapper):
>>>     cur_log = {}
>>>     y_pred = wrapper.predict(self.X).argmax(axis=1)
>>>     acc = accuracy_score(y_true=self.y, y_pred=y_pred)
>>>     cur_log["epoch"] = wrapper.cur_epoch
>>>     cur_log["acc"] = acc
>>>     self.logs.append(cur_log)
>>>     if self.verbose > 0:
>>>         print("[Epoch {:5d}] Test ACC: {:.4f}".format(cur_log["epoch"], cur_log[
↪"acc"]))
```

Parameters **wrapper** (Wrapper) – The training *Wrapper*.

class EarlyStoppingACC(*X*, *y*, *patience=1*, *verbose=0*, *save_path=None*)

Early-stopping by classification accuracy.

Parameters

- **X** (*numpy.array*) – Feature matrix with the size of $[N, D]$.
- **y** (*numpy.array*) – Label matrix with the size of $[N, 1]$.
- **patience** (*int*) – Number of epochs with no improvement after which training will be stopped.
- **verbose** (*int*) – verbosity mode.
- **save_path** (*str*) – If `save_path=None`, do not save models, else save the model with the best accuracy to the given path.

on_epoch_end(*self*, *wrapper*)

Calculate the validation accuracy and determine whether to stop training.

```
>>> def on_epoch_end(self, wrapper):
>>>     cur_log = {}
>>>     y_pred = wrapper.predict(self.X).argmax(axis=1)
>>>     acc = accuracy_score(y_true=self.y, y_pred=y_pred)
>>>     if acc > self.best_acc:
>>>         self.best_acc = acc
>>>         self.cnt = 0
>>>         if self.save_path is not None:
>>>             wrapper.save(self.save_path)
>>>     else:
>>>         self.cnt += 1
>>>         if self.cnt > self.patience:
>>>              wrapper.stop_training = True
>>>     cur_log["epoch"] = wrapper.cur_epoch
>>>     cur_log["acc"] = acc
>>>     cur_log["best_acc"] = self.best_acc
>>>     self.logs.append(cur_log)
>>>     if self.verbose > 0:
>>>         print("[Epoch {:5d}] EarlyStopping Callback ACC: {:.4f}, Best ACC:
↪{:.4f}".format(cur_log["epoch"], cur_log["acc"], cur_log["best_acc"]))
```

Parameters **wrapper** (`Wrapper`) – The training *`Wrapper`*.

5.5 pytsk.gradient_descent.utils

check_tensor(*tensor*, *dtype*)

Convert `tensor` into a `dtype` torch.Tensor.

Parameters

- **tensor** (*`numpy.array/torch.tensor`*) – Input data.
- **dtype** (*`str`*) – PyTorch dtype string.

Returns A `dtype` torch.Tensor.

reset_params(*model*)

Reset all parameters in `model`.

Parameters **model** (*`torch.nn.Module`*) – Pytorch model.

class NumpyDataLoader(**inputs*)

Convert numpy arrays into a dataloader.

Parameters **inputs** (*`numpy.array`*) – Numpy arrays.

INDEX

T

U

W

X