DSIPTS: unified library for timeseries modelling

This library allows to: 1- load timeseries in a convenient format 2- create tool timeseries with controlled categorical features (additive and multiplicative) 3- load public timeseries 4- train a predictive model using different pytroch architectures

Background

Let $X(t)$ be a multivariate timeseries, e.g. $\forall t, X(t) \in \mathbf{R}^k$ for some k. The vector space $\mathbf{R}^k$ can be partitioned into two disjoint sets: the categorical features $\mathcal{C} \subset \mathbf{N}^c$ and continuous features $\mathcal{W} \subset \mathbf{R}^{k-c}$. We assume that $\mathcal{C}$ is known for each t. Let $\mathcal{F} \subset \mathbf{R}^f$ be the set of known variables for each t, $\mathcal{P} \subset \mathbf{R}^p$ be the set of variables known until time t, and $\mathcal{T} \subset \mathcal{P} \subset \mathbf{R}^s$ the target variables. Let also define $\tau \in N$ as the number of lag for wich we want a forecast, then the aim of a predictive model is to find a function $F : \mathbf{R}^k \to \mathbf{R}^{s \times \tau}$ such as:

$$F(\mathcal{C}(t-K,\dots,t+\tau),\mathcal{F}(t-K,\dots,t+\tau),\mathcal{P}(t-K,\dots,t),\mathcal{T}(t-K,\dots,t)) = \mathcal{T}(t+1,\dots,t+\tau)$$

for some K representing the maximum past context.

In the library we adopt some convention that must be used when developing a new model:

```
y : the target variable(s)
x_num_past: the numerical past variables
x_num_future: the numerical future variables
x_cat_past: the categorical past variables
x_cat_future: the categorical future variables
idx_target: index containing the y variables in the past dataset. Can be used during the training for train a differential model
```

by default, during the dataset construction, the target variable will be added to the `x_num_past` list. Moreover the set of categorical variable will be the same in the past and the future but we choose to distinguish the two parts during the forward loop for seek of generability.

During the forward process, the batch is a dictionary with some of the key showed above, remember that not all keys are always present (check it please) and build a model accordlying. The shape of such tensor are in the form $[B, L, C]$ where B indicates the batch size, L the length and C the number of channels.

The output of a new model must be $[B, L, C, 1]$ in case of single prediction or $[B, L, C, 3]$ in case you are using quantile loss.

Try to reuse some of the common keyworks while building your model. After the initialization of the model you can use whatever variable you want but during the initialization please use the following conventions. This frist block maybe is common between several architectures:

- **past_steps** = int. THIS IS CRUCIAL and self explanatory
- **future_steps** = int. THIS IS CRUCIAL and self explanatory
- **past_channels** = len(ts.num_var). THIS IS CRUCIAL and self explanatory
- **future_channels** = len(ts.future_variables). THIS IS CRUCIAL and self explanatory
- **embs** = [ts.dataset[c].nunique() for c in ts.cat_var]. THIS IS CRUCIAL and self explanatory.
- **out_channels** = len(ts.target_variables). THIS IS CRUCIAL and self explanatory
- **cat_emb_dim** = int. Dimension of embedded categorical variables, the choice here is to use a constant value and let the user chose if concatenate or sum the variables
- **sum_emb** = boolean. If true the contribution of each categorical variable is summed
- **quantiles**=[0.1,0.5,0.9]. Quantiles for quantile loss
- **kind** =str. If there are some similar architectures with small differences maybe is better to use the same code specifying some properties (e.g. GRU vs LSTM)
- **activation**= str ('torch.nn.ReLU' default). activation function between layers (see pytorch activation functions)
- **optim**= str ('torch.optim.Adam' default). optimization function (see pytorch optimization functions
- **dropout_rate**=float (0.1 default). dropout rate
- **use_bn**=boolean (False default). Use or not batch normalization
- **persistence_weight**= float (0.2 default). Penalization weight for persistent predictions
- **loss_type**= str (mse default)

some are more specific for RNN-CONV architectures:

- **hidden_RNN** = int. If there are some RNN use this and the following
- **num_layers_RNN** = int.
- **kernel_size** = int. If there are some convolutional layers

linear: - **hidden_size** = int. Usually the hidden dimension, for some architecture maybe you can pass the list of the dimensions - **kind** =str. Type of linear approach

or attention based models:

- **d_model** = int .d_model of a typical attention layer
- **num_heads** = int .Heads
- **dropout** = float. dropout
- **n_layer_encoder** = int. encoder layers
- **n_layer_decoder** = int. decoder layers

How to

In a pre-generated environment install pytorch and pytorch-lightning (`pip install pytorch-lightning==1.9.4`) then go inside the lib folder and execute:

```
python setup_local.py install --force
```

Alternatively, you can install it from the package registry:

```
pip install --force dsipts --index-url https://dsipts:glpat-98SR11neR7hzxy__SueG@gitlab.fbk.eu/api/v4/projects/4571/packages/pypi
```

In this case it will installed the last version submitted to the package registry. For testing pourpose please use the first method (in local). For using the latter method ask to `agobbi@fbk.eu`.

Test

You can test your model using a tool timeseries

```
##import modules
from dsipts import Categorical,TimeSeries, RNN, Attention

#############define some categorical features##########

##weekly, multiplicative
settimana = Categorical('settimanale',1,[1,1,1,1,1,1,1],7,'multiplicative',[0.9,0.8,0.7,0.6,0.5,0.99,0.99])

##montly, additive (here there are only 5 month)
mese = Categorical('mensile',1,[31,28,20,10,33],5,'additive',[10,20,-10,20,0])

##spot categorical variables: in this case it occurs every 100 days and it lasts 7 days adding 10 to the original timeseries
spot = Categorical('spot',100,[7],1,'additive',[10])

##initizate a timeseries object
ts = TimeSeries('prova')
```

The baseline tool timeseries is defined as:

$$y(t) = \left[A(t) + 10\cos\left(\frac{50}{l \cdot \pi}\right)\right] * M(t) + Noise$$

where l is the length of the signal, $A(t)$ correspond to all the contribution of the additive categorical variable and $M(t)$ all the multiplicative contributions.

We can now generate a timeseries of length 5000 and the cateorical features described above:

```
ts.generate_signal(noise_mean=1,categorical_variables=[settimana,mese,spot],length=5000,type=0)
```

`type=0` is the base function used. In this case the name of the time variable is `time` and the timeseries is called `signal`.

In a real application generally we have a pandas data frame with a temporal column and a set of numerical/categorical features. In this case we can define a timeseries object as:

```
ts.load_signal(dataset,past_variables =[list of past variables],target_variables =[list of target variables],cat_var = [categoric
```

Up to now, the automathic categorical features extracted can be: `'hour','dow','month','minute'`. If you want to use a public dataset there is a wrapper in the library for downloading some datasets using Monarch.

```
from dsipts Monarch
import pandas as pd
m = Monarch(filename='monarch',baseUrl='https://forecastingdata.org/', rebuild=True)
```

This code will scrap the website and save the URLs connected to the dataset. After downloading it will save a file using the `filename` and, the next time you use it you can set `rebuild=False` avoinding the scraping procedure. After that `m.table` contains the table. Each dataset has an ID, you can downloadthe data:

```
m.download_dataset(path='data',id=4656144,rebuild=True)
m.save()##remember to save after each download in order to update the class for following uses.
```

In the attribute `m.downloaded` you can see a dictionary with an association ID–> folder. Finally, you can get the data from the downloaded files:

```
loaded_data,frequency,forecast_horizon,contain_missing_values,contain_equal_length = m.generate_dataset(4656144)
```

and create a timeseries object using the auxiliary function `get_freq`:

```
serie = pd.DataFrame({'signal':loaded_data.series_value.iloc[0]})
serie['time'] = pd.date_range(start = loaded_data.start_timestamp.iloc[0], periods=  serie.shape[0],freq=get_freq(frequency))
serie['cum'] = serie.time.dt.minute  + serie.time.dt.hour
starting_point = {'cum':0} ##this can be used for creating the dataset: only samples with cum=0 in the first future lag will be u
ts = TimeSeries('4656144')
ts.load_signal(serie.iloc[0:8000],enrich_cat=['dow','hour'],target_variables=['signal'])
```

Now we can define a forecasting problem, for example using the last 100 steps for predicting the 20 steps in the future. In this case we have one time series so:

```
past_steps = 100
future_steps = 20
```

Let suppose to use a RNN encoder-decoder sturcture, then the model has the following parameters:

```
config = dict(model_configs =dict(
                                    cat_emb_dim = 16,
                                    hidden_LSTM = 256,
                                    num_layers_LSTM = 2,
                                    sum_emb = True,
                                    kernel_size_encoder = 20,
                                    past_steps = past_steps,
                                    future_steps = future_steps,
                                    past_channels = len(ts.num_var),
                                    future_channels = len(ts.future_variables),
                                    embs = [ts.dataset[c].nunique() for c in ts.cat_var],
                                    quantiles=[0.1,0.5,0.9],
                                    use_bn = False,
                                    activation='selu',
                                    out_channels = len(ts.target_variables),
                scheduler_config = dict(gamma=0.1,step_size=100),
                optim_config = dict(lr = 0.0005,weight_decay=0.01))
model_sum = RNN(**config['model_configs'],optim_config = config['optim_config'],scheduler_config =config['scheduler_config'] )
ts.set_model(model_sum,config=config )
```

Notice that there are some free parameters: `cat_emb_dim` for example represent the dimension of the embedded categorical variable, `sum_embs` will sum all the categorical contribution otherwise it will concatenate them. It is possible to use a quantile loss, specify some parameters of the scheduler (StepLR) and optimizer parameters (Adam).

Now we are ready to split and train our model using:

```
ts.train_model(dirpath=<path to weights>,split_params=dict(perc_train=0.6, perc_valid=0.2,past_steps = past_steps,future_steps=fu
```

It is possble to split the data indicating the percentage of data to use in train, validation, test or the ranges. The `shift` parameters indicates if there is a shift constucting the y array. It is used for the attention model where we need to know the first value of the timeseries to predict. The `skip_step` parameters indicates how many temporal steps there are between samples. If you need a futture signal that is long `skip_step+future_steps` then you should put `keep_entire_seq_while_shifting` to True (see Informer model).

During the training phase a log stream will be generated. If a single process is spawned the log will be displayed, otherwise (see `batch_example`) a file will be generated. Moreover, inside the `weight` path there wil be the `loss.csv` file containing the running losses.

Models

A description of each model can be found in the class documentation here

Usage

In the folder `bash_examples` you can find an example in wich the library is used for training a model from command line using OmegaConf and Hydra with more updated models and examples.

TODO

- add more sintetic data
- more versatility during the data loading process

Documentation

You can find the documentation here: or in the folder `docs/_build/html/index.html` If yon need to generate the documentation after some modification just run:

```
./make_doc.sh
```

and add the new files to the git repo.

For user only: see ci file and enable public pages

Adding new models

If you want to add a model:

- extend the `Base` class in `dsipts/models`
- add the export line in the `dsipts/__init__.py`
- add a full configuration file in `bash_examples/config_test/architecture`