Usage of DSIPTS

This repo collect some examples related to the use of dsipts. Using this repo you can train some DL models for timeseries forecasting using public datasets like Monash or six_dataset used for benchmarking timeseries models. It is possible to use also the Venice dataset described here.

Here we report a complete approach using Hydra that allows to better manage multiple experiments.

This particular repo is structured for working with the six datasets (that are 9 because ethh has different resolution and periods): `['electricity','etth1','etth2','ettm1','ettm2','exchange_rate','illness','traffic','weather']`. The task is to predict `y` using past data and other past covariates for a variable number of steps depending on the paper used as reference.

Installation

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
```

Configuration

- copy the folder `all_six_datasets` inside a data folder (in what follows `/home/agobbi/Projects/ExpTS/data`).
- place yoursel in `bash_examples`
- train the models
- create the folders `csv` and `plots` in the `pathdir` in my case `/home/agobbi/Projects/ExpTS`

Hydra

In you environment install hydra and the joblib launcher for the paralellization tasks:

```
pip install hydra-core
pip install hydra-joblib-launcher      ## if you have a big gpu
pip install hydra-submitit-launcher    ## if you are in a slurm envirionment
```

```
pip install hydra-optuna-sweeper       ## if you need optuna
```

The script used are: - **train.py** for training models - **inference.py** for inference - **compare.py** for comparing different models

This structure is a convient way to deal with multiple experiments, feel free to adjust it as you prefere. There are some trick for extracting runtime the hydra choices (and use informative names for the models). This can be ugly to see but it easy to compare the same model with different parameters. If you want to use you own data with this schema you need to add your data processing pipeline in `lodad_data` and define your own timeseries object. For example in the follwing snippet we have 3 continuous variables: `Value, rain temp` that are assumed to be known also in the future while predicting `Value`. The month column will be created as categorical feature.

```
    ts.load_signal(data_ex,past_variables =['Value','rain','temp'],future_variables = ['rain','temp'],target_variables =['Value']
```

Remember also to modify also `train` and `inference` accordingly to your implemented function.

Hydra is used for composing configuration files. In our case most of the parameter can be reused among the different models and are collected under the general configuration file `config/config.yaml`. In what follows the `weather` dataset is used, and notice that this dataset has a frequency of **10 minutes**. The parameters here are the same described in the `dsitps` documentation but clearly some of them can not be modified since they depend on the selected time series. The configuration files related to this experiment can be found in `config_weather`; a generic config folder contains:

```
config_gpu.yaml              # containing the global configuration usually for gpu or local train, see below one example
config_slurm.yaml            # containing the global configuration for slurm training see below one example
compare.yaml                 # instructions for comparing different models
architecture/                # the folder containing the configurations specific for all the models to test
config_used/                 # this folder will be populated while training the models, and will be used in the comparison phase
```

The config file in the case of the weather dataset is reported and commented below.

```
dataset:
  dataset: 'weather'
  path: '/home/agobbi/Projects/ExpTS/data' ##path to data. In the folder data must be present the folder six_dataset

scheduler_config:
  gamma: 0.75
  step_size: 250

optim_config:
  lr: 0.0005
  weight_decay: 0.001
```

```yaml
model_configs:
  past_steps: 16
  future_steps: 16
  quantiles: [0.1,0.5,0.9]  #if you want to use quantile loss, otherwise set it to []
  past_channels : null      #dataset dependent  hydra expect you to set it anyway also if it depends on data
  future_channels : null    #dataset dependent
  embs: null                #dataset dependent
  out_channels: null        #dataset dependent

split_params:
  perc_train: 0.8
  perc_valid: 0.1
  range_train: null
  range_validation: null
  range_test: null
  shift: 0
  starting_point: null
  skip_step: 1
  past_steps: model_configs@past_steps          #this is a convinient what to reuse previous information, thx omegaconf
  future_steps: model_configs@future_steps

train_config:
  dirpath: "/home/agobbi/Projects/ExpTS"
  num_workers: 0
  auto_lr_find: true   ##this allows to pytorch lightening to find a suitable lr
  devices: [0]

inference:
  output_path: "/home/agobbi/Projects/ExpTS"
  load_last: true
  batch_size: 200
  num_workers: 4
  set: "validation"
  rescaling: false  #(sometimes you want to get the errors on normalized datasets)
```

Train

After declaring some stuff about the dataset and the enviroment we can train the model(s). Depending on the architecture (slurm or gpu/cpu) you need to add the following blocks to the main config:

For gpu/cpu:

```
defaults:
  - _self_                            # take all this configuration
  - architecture: null                # and let the use specify the architecture to use (be aware that the filed here is the same
  - override hydra/launcher: joblib  # use joblib for multiprocess allowing parallelization in case of multirun in a gpu/cpu envi


hydra:
  launcher:
    n_jobs: 2                   # parameters indicate the number of parallel jobs in case of multirun
    batch_size:2                #2 parralel train session
    pre_dispatch: 4
    _target_: hydra_plugins.hydra_joblib_launcher.joblib_launcher.JoblibLauncher

  output_subdir: null          # do not save any file
  sweeper:
    params:
      architecture: glob(*)    # this is a way to train all the models in the architecure folder
```

For a SLURM cluster:

```
defaults:
  - _self_
  - architecture: null
  - override hydra/launcher: submitit_slurm  ## use slurm launcher

hydra:
  launcher:
    submitit_folder: ${hydra.sweep.dir}/.submitit/%j
    timeout_min: 6000
    partition: gpu-V100      ##partition to use REQURED
    mem_gb: 6                ##gb requires      REQURED
```

```
    nodes: 1
    gres: gpu:1               ##number of GPU    REQURED
    array_parallelism: 10     ##parallel process

    name: ${hydra.job.name}
    _target_: hydra_plugins.hydra_submitit_launcher.submitit_launcher.SlurmLauncher
    setup:
      - conda activate tt  ##activate the conda environment first!

    output_subdir: null
    sweeper:
    params:
      architecture: glob(*)
```

In the `config_weather/architecture` folder there are the selected models that have the following structure:

```
# @package _global_  ##care this must be present!


#the specified parameters below overwrite the default configuration having a more compact representation
model:
  type: 'linear'
  retrain: true  ## overwrite the model with the same parameters


ts:
  name: 'weather'
  version: 1              # if you need to versioning a model
  enrich: ['hour']
  use_covariates: false   # if true all the columns of the dataset will be used as past features


## for more information about models please look at the documentation [here] (https://dsip.pages.fbk.eu/dsip_dlresearch/timeserie
model_configs:
  cat_emb_dim: 32              # dimension of categorical variables
  kernel_size: 5               # kernel size
  sum_emb: true                # if true each embdedding will be summed otherwise stacked
```

```
  hidden_size: 256            # hidden size of the fully connected block
  kind: 'linear'              # model type
  dropout_rate: 0.2           # dropout
  use_bn: false               # use or not bn layers in the first layers
  optim: torch.optim.Adam     # optimizer
  activation: torch.nn.PReLU  # activation between linear
  persistence_weight: 0.010   # in case of loss different from l1 or mse it is used to weight a penality score
  loss_type: 'l1'             # loss



train_config:
  batch_size: 128
  max_epochs: 250
  gradient_clip_val: null          # pytorch lightening gradient clipping procedure
  gradient_clip_algorithm: 'norm'  # pytorch lightening gradient clipping procedure
```

#########FINO A QUIEEEEEEEEEEEEEE Hydra allows us to train a specific model using if you are in a gpu environment

```
python train.py  architecture=linear --config-dir=config_weather --config-name=config_gpu
```

or, if you are in a slurm gpu cluster

```
python train.py  architecture=linear --config-dir=config_weather --config-name=config_slurm -m
```

The `-m` option is important because generally we would't lauch the script in the frontend. This shortcut allows hydra to use the multirun scheduler and lauch the process(es) in the required nodes. In the case of a single gpu machine it will lauch parallel process (careful to the used VRAM).

For example we can train two models on the same data using:

```
python train.py  -m architecture=linear, dlinear --config-dir=config_weather --config-name=config_slurm
```

or all the implemented models:

```
python train.py  -m  --config-dir=config_weather --config-name=config_slurm
```

this because there is the option

```
    params:
```

```
architecture: glob(*)
```

that compile the config files using all the models declared in the architecture folder.

In case of parallel experiment you should see at display something like:

```
(tt) agobbi@frontend:~/Projects/timeseries/bash_examples$ python train.py --config-dir=config_weather --config-name=config_slurm -m
[2023-05-03 16:46:59,679][HYDRA] Submitit 'slurm' sweep output dir : multirun/2023-05-03/16-46-58
[2023-05-03 16:46:59,680][HYDRA]        #0 : architecture=attention
[2023-05-03 16:46:59,690][HYDRA]        #1 : architecture=d3vae
[2023-05-03 16:46:59,696][HYDRA]        #2 : architecture=dlinear
[2023-05-03 16:46:59,700][HYDRA]        #3 : architecture=gru
[2023-05-03 16:46:59,710][HYDRA]        #4 : architecture=informer
[2023-05-03 16:46:59,715][HYDRA]        #5 : architecture=linear
[2023-05-03 16:46:59,720][HYDRA]        #6 : architecture=lstm
[2023-05-03 16:46:59,733][HYDRA]        #7 : architecture=mymodel
[2023-05-03 16:46:59,738][HYDRA]        #8 : architecture=mymodel_v2
[2023-05-03 16:46:59,742][HYDRA]        #9 : architecture=mymodel_v3
[2023-05-03 16:46:59,747][HYDRA]        #10 : architecture=nlinear
[2023-05-03 16:46:59,757][HYDRA]        #11 : architecture=persistent
```

Hydra will create a folder called `multirun` with all the experiments lauched nested as `date/time/x` where `x` indicates the id of the lauched joib. Inside `date/time/x` there will be a file called `train.log` containing all the information logged by the system and useful for debugging. Moreover, the error message and output messages generated by the `sbatch` slurm command can be found in the `date/time/.submitted/x` folder.

If the row `override hydra/launcher: joblib` is commented the train will be consecutive, otherwise in parallel. In the latter case the output in the terminal will be a mess but in the multirun folder you can find all the log files in the folder `0,1,...` based on the number of models you are training.

Last but not least: you can lauch as many process as you want but only few will be active at the same time: if slur is used the keyword is `array_parallelism: 10` otherwise the combination of `batch_size:2` and `n_jobs: 2` for the single gpu pipeline.

Inference

Once the models are trained, the relative full configurations are saved in `config_used` and can be used for inference or comparison:

```
python compare.py --config-dir=config_weather --config-name=compare
```

where the compare file is:

```
models: 'config_weather'               ## path to the main config folder or list of configuration files
dirpath: "/home/agobbi/Projects/ExpTS" ## where are store the models and where to put the results
```

```
set: 'test'                              ## set to test
name: 'prova'
rescaling: false                         ## sometimes want to get the MSE on the scaled data
batch_size: 32                           ## batch size for the dataloader
```

or if you are in a slurm cluster remembrer to add the `-m` parameters also for the comparison step (otherwise the inference will be execute in the frontend)

```
python compare.py --config-dir=config_weather --config-name=compare_slurm -m
```

if you are in a SLURM cluster where che `compare_slurm.yaml` file must contain also something like:

```
defaults:
  - _self_
  - override hydra/launcher: submitit_slurm

hydra:
  launcher:
    submitit_folder: ${hydra.sweep.dir}/.submitit/%j
    timeout_min: 600
    partition: gpu-V100
    mem_gb: 6
    nodes: 1
    gres: gpu:1
    name: ${hydra.job.name}
    _target_: hydra_plugins.hydra_submitit_launcher.submitit_launcher.SlurmLauncher
    setup:
      - conda activate tt
```

In the `dirpath` folder `/home/agobbi/Projects/ExpTS/` there are three folder now: `weights` containing the model and the weights, `plots` containing some plots coming from the `compare` script and the `csv` forder containing the files.

A typical example of plot is displayed below and shows the MSE at different lags in the test set for different models:

The loss plot is currenty broken on server, you can reproduce it form the notebook `4- results` (see the notebook section)

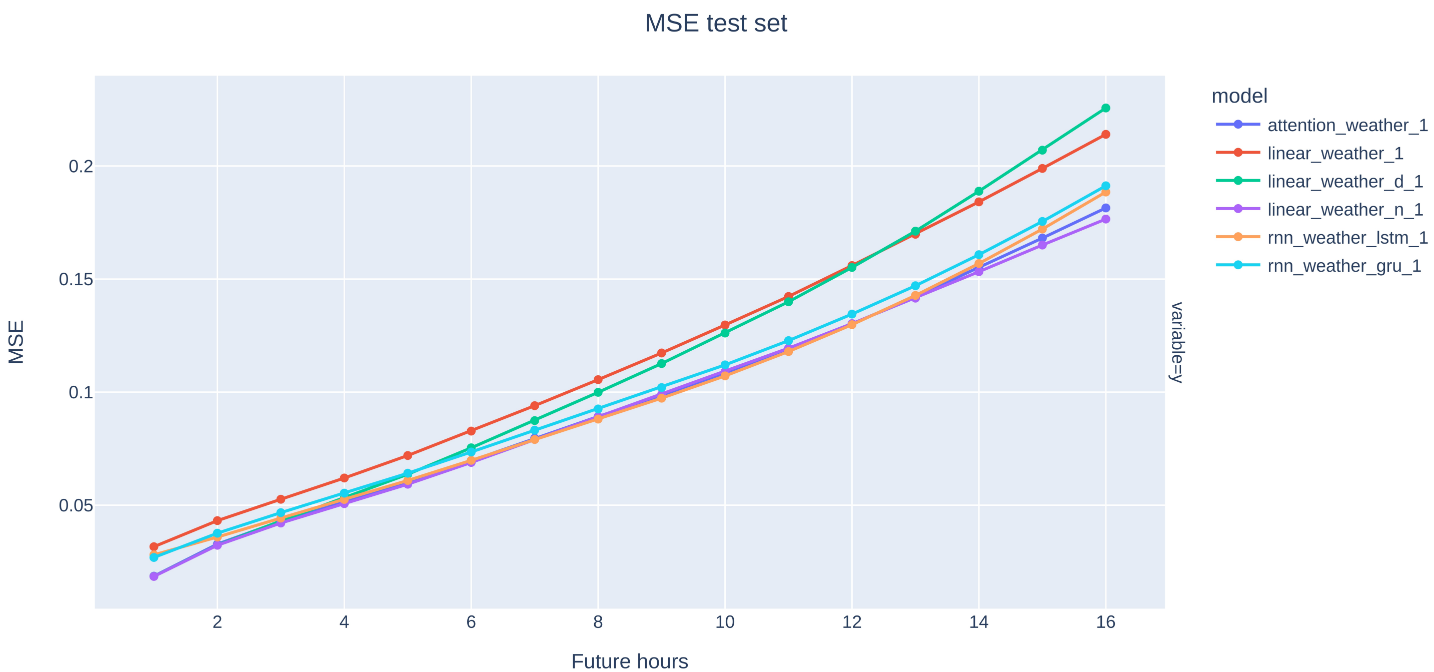

Figure 1: plot

Testing

You can use the `config_test` for testing your models. In this case you can use smaller model with fewer epochs:

```
python train.py -m --config-dir=config_test --config-name=config_gpu
```

Same model differtent parameters

It is possible also to perform a fine tuning procedure on a specific model, in this case:

```
python train.py --config-dir=config_weather --config-name=config_slurm -m architecture=linear model_configs.hidden_RNN=32,64,128
```

will spawn 3 paralle process trainin the same model with three different values of `hidden_RNN`. In case of multiple parameters to test hydra will generate all the couples of possibilities. This approach can explode very quickly, for this reason it is possible to use `optuna` for exploring the space of the configurations (THIS FEATURE IS NOT MATURE):

```
python train.py --config-dir=config_weather --config-name=config_slurm_optuna -m
```

In this case the configuration file should include the following part:

```
defaults:
  - _self_
  - architecture: null
  - override hydra/launcher: submitit_slurm #SLURM STUFF
  - override hydra/sweeper: optuna          ##OPTUNA SWEEPER

hydra:  ##SLURM STUFFS
  launcher:
    submitit_folder: ${hydra.sweep.dir}/.submitit/%j
    timeout_min: 600
    partition: gpu-V100
    mem_gb: 6
    nodes: 1
    gres: gpu:1
    name: ${hydra.job.name}
    _target_: hydra_plugins.hydra_submitit_launcher.submitit_launcher.SlurmLauncher
    setup:
      - conda activate tt
```

```
  output_subdir: null
  sweeper:
    sampler: ##OPTUNA STUFFS
      _target_: optuna.samplers.TPESampler ## see https://optuna.readthedocs.io/en/stable/reference/samplers/index.html for other
      seed: 123
      consider_prior: true
      prior_weight: 1.0
      consider_magic_clip: true
      consider_endpoints: false
      n_startup_trials: 10
      n_ei_candidates: 24
      multivariate: false
      warn_independent_sampling: true
    _target_: hydra_plugins.hydra_optuna_sweeper.optuna_sweeper.OptunaSweeper
    direction: minimize
    storage: sqlite:///tutorial.db #or null
    study_name: tft
    n_trials: 3 ## put the number of maximun trials
    n_jobs: 3   ## parallel jobs


    params:
      architecture: linear                                  #architecture you want to finetune
      model_configs.num_layers_RNN: choice(1,2,3)           #parameters you want to explore, categorical
      model_configs.persistence_weight: range(0.1,0.9,0.1) #or continuous
```

The best configuaration can be found in `multirun/<DATE/<TIME>/optimization_results.yaml` and it show something like:

```
name: optuna
best_params:
  model_configs.cat_emb_dim: 2
  model_configs.dropout_rate: 0.7
best_value: 16.59403419494629
```

where `best_value` is the mean loss in the validation step and `best_parameters` contains the best configuration. Pay attention and use the same loss for all the experiments, some losses have a different scale and can not compare!

If `storage:sqlite:///<DBNAME>.db` a sqlit file fill be created listing all the trials performed and can easily recovered using

```
df = pd.read_sql_query("SELECT  t.trial_id,param_name, param_value,value FROM trial_params as t join trial_values as v on t.trial
```

This dataframe contains the parameters tested

```
from copy import deepcopy
def dict_of_dicts_merge(x, y):
    z = {}
    overlapping_keys = x.keys() & y.keys()
    for key in overlapping_keys:
        z[key] = dict_of_dicts_merge(x[key], y[key])
    for key in x.keys() - overlapping_keys:
        z[key] = deepcopy(x[key])
    for key in y.keys() - overlapping_keys:
        z[key] = deepcopy(y[key])
    return z

def unroll(x):
  result = {}
  for index, row in x.iterrows():
    keys = row['param_name'].split('.')
    tmp = {keys[-1]:row['param_value']}
    for i in range(len(keys)-1):
      tmp = {keys[-i-2]:tmp}
    result = dict_of_dicts_merge(result, tmp)
    #import pdb
    #pdb.set_trace()
  return pd.Series({'config':result,'loss':row['value']})

df.groupby('trial_id').apply(unroll).reset_index()
```

Results

Here you can find some plots comparing different models.

Tips

- The folder `weights` can become very big. Try to remove all useless experiments.
- The folder `multirun` can become very crowded, not so big but it can be deleted sometimes
- The folder `config_used` contains all the trained models. The comparison step can be very time consuming (the models are evaluated sequentially). Please use only the models you need
- Do not mix loss function using the optuna sweeper

Stack generalization

Once you have trained a bunch of good models you may try to train a stack generalization model: a (usually simpler) model that combines the output of the trained model and estimate the target. You can use the routine `train_stack.py` similarly to `train.py` but with some differences: - the config file for the stacked model is in the folder `stack` - you need to add `stacked=True` when creating the timeseries object - you need to add a section called `stack` similar to the snipped below to the configuration file - if you want to a special launcher (joblib or slurm) you need a little workaround

```
stack:
  #models: config_test  ## if you want to use all the models
  models: ['config_incube/config_used/tft2_test_1_loss_type=std_normpersistence_weight=1.yaml','config_incube/config_used/crossfo
  dirpath: "/home/agobbi/Projects/ExpTS/incube"
  set: 'validation' ## the dataset of the trained model to use as a training set
  name: 'prova'
  rescaling: true
  batch_size: 64
```

the splitting section of the stacked model is referred to the subset defined in the `stack` section. If you use the `validation` set from the original model and set a training percentage of 0.8 the stacked model is trained on the 80% of the validation set.

Finally you can train your model using:

```
python train_stack.py  --config-dir=config_test --config-name=config stack=linear architecture=stack hydra.launcher.n_jobs=1 -m
```

here you can see the workaround `architecture=stack` (you need to create an empty file in the `architecture` folder) and set `hydra.launcher.n_jobs=1` otherwise the multiprocess function raises some errors. If you can run the code without sweeper you can simple run:

```
python train_stack.py  --config-dir=config_test --config-name=config stack=linear
```

When you lauch `compare.py` magically it will run also the stacked model and add it to the model's pool (very cool, isn't it?).

Groups

If you have more entities in your dataset (for example pollen stations, facilities) that share the same datastream you can use a model for each entity or a single model trained using all the data available. You can simply add the `group` parameter while defining the timeseries:

```
ts.load_signal(data,past_variables = meteo + target,
               future_variables = meteo,
               target_variables =target,
               cat_var= cat_var+[group],
               enrich_cat= conf.ts.enrich,
               group=group)
```

Only percentage split is allowed and the splitting procedure is performed for each entity independetely. The temporal split is allowed but on your risk (it is performed globally). During the splitting procedure you can choose to normalize the dataset for each entity using the parameter `normalize_per_group=True`. In this case both categorical and numerical features are normalized independetely for each entity. The `group` column is added to the categorical variables pool available during the training process, please use a model that can deal with categorical variables. In the inference step the column `group` will be added to the final dataset.

Silly model

If you need to test your architecture, expecially the decoder part that uses `x_num_future` as input, it is possible to include the target values as input using `silly=True` in the load function. This allow to see if all the pipeline works properly (remember to remove it when training the final model).