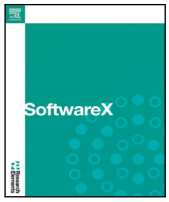
SoftwareX 28 (2024) 101875

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

DSIPTS: A high productivity environment for time series forecasting models

Andrea Gobbi *, Andrea Martinelli, Marco Cristoforetti

Fondazione Bruno Kessler, Via Sommarive 18, Trento, 38121, Italy

ARTICLE INFO

Keywords:
Time series forecasting
Deep learning

ABSTRACT

Several Python libraries have been released for training time series forecasting models in the last few years. Most include classical statistical approaches, machine learning models, and recent deep learning architectures. Despite the great work for releasing such open-source resources, a tool that allows testing Deep Learning architectures in a framework that guarantees transparent input output management, reproducibility of the results, and expandability of the supported models is still lacking. With DSIPTS, we fill this gap, providing the community with a tool for training and comparing Deep Learning models in the time series forecasting field.

Code metadata

Current code version	1.1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00056
Permanent link to Reproducible Capsule	
Legal Code License	Apache v2.0
Code versioning system used	git
Software code languages, tools, and services used	python
Compilation requirements, operating environments & dependencies	pandas, scikit-learn, Omegaconf, plotly, sphinx, sphinx_rtd_theme, hydra-core, pytorch-lightning, hydra-optuna-sweeper, hydra-joblib-launcher, hydra-submitit-launcher, torch, aim
If available Link to developer documentation/manual	https://dsip.pages.fbk.eu/dsip_dlresearch/timeseries/
Support email for questions	agobbi@fbk.eu

1. Motivation and significance

A unified, fast, and solid framework to test different deep learning architectures for multivariate time series forecasting can improve performance comparison and shorten the deployment time. Along this line, we propose DSIPTS: a Python library for training deep learning forecasting models for time series. Leveraging the power of PyTorch Lightning [1] and Hydra [2], DSIPTS offers a configurable solution adaptable also to SLURM cluster environments and, in general, CPU/GPU systems. The library can manage complex scenarios, specifically addressing multivariate-multistep grouped time series and covariate definition. Moreover, DSIPTS facilitates the efficient reuse of fitted models, empowering the creation of versatile generalized stacked models.

With DSIPTS, the user can define a time series object from a pandas dataset, specifying the target variables and all the covariates (numerical and categorical, past and future), and train and compare different Deep Learning (DL) models, minimizing one from the available loss functions. It is possible to select one among the implemented architectures or extend the base class following only a few constraints to be compatible with the batch generation routine and training/inference procedure. Finally, it is also possible to load benchmark time series from freely available datasets [3–5] for testing deep learning architectures.

An exhaustive list of machine learning and deep learning models, libraries, tutorials, and notebooks on time series forecasts can be found in [6]. In what follows, we briefly describe some python libraries similar to DSIPTS.

Darts [7] contains several statistical models, machine learning algorithms, and DL architectures for time series forecasting and anomaly detection. Well-documented notebooks allow the user to explore the library easily. It can handle past and future covariates and it can easily integrate ray tune [8] for parameter optimization. Still, it lacks native

* Corresponding author.
E-mail address: agobbi@fbk.eu (Andrea Gobbi).

https://doi.org/10.1016/j.softx.2024.101875
Received 31 January 2024; Received in revised form 20 June 2024; Accepted 1 September 2024
Available online 16 September 2024

support for massive training and more recent deep learning architectures. Moreover, during the inference phase, the predicted values seem not to deterministic, and there is no reference about the forecasting future step. These lags are useful for estimating how far the model is effective in forecasting with a desired error level. Finally, the batch structure is not fixed among the deep-learning architecture, which can cause issues when implementing a custom architecture. PyTorch Forecasting [9] implements six deep learning architectures with connection with the Optuna library [10] for hyperparameter tuning and with tools for inspecting the correlation between input and output exploring the interpretative attention mechanism of the Temporal Fusion Transformer (TFT) model [11]. Unfortunately, it is outdated, and it does not contain the most recently released architectures. The Python library sktime [12] contains a huge set of machine learning models using scikit-learn-like syntax but, also in this case, it does not contain recent deep learning architectures.. Facebook released two software: Kats [13] for time series analysis, and Prophet [14] for forecasting. This last with particular attention on multiple seasonality patterns and providing confidence intervals on the forecasts. Despite AutoTS [15] seems a promising solution for developing fast and accurate time series models with non-conventional loss functions and ensemble models, it lacks internally recent DL network architectures that are instead inherited from GluonTS. GluonTS [16] is a probabilistic deep-learning-based library for creating fully customized architectures in a mature ecosystem. Unfortunately, it contains only a few DL modules mainly implemented in MXNET [17].

2. Software description

The idea of DSIPTS comes from the need for an environment where training and serving forecasting models with features oriented both for research purposes and in-production scenarios. The three paradigms adopted in developing DSIPTS are transparency, reproducibility, and expandability.

- Transparency in the input–output management: in DSIPTS the output generation process is clear and easy to understand.
- Reproducibility to guarantee the quality and integrity of the results and understand the impact of random processes in the training procedure.
- Expandability allows the user to develop a custom DL model, simplifying the testing procedure and providing a simple interface for comparisons with other architectures.

In DSIPTS, models are implemented in PyTorch and trained using PyTorch Lightning. Data are managed using pandas [18] and numpy [19] and preprocessed using sklearn [20]. DSIPTS is paired with an environment called `bash_examples` for managing multiple models based on Hydra [2], a framework for managing nested configurations. In what follows, we will describe the main features of DSIPTS. Python classes are highlighted in bold.

2.1. Software architecture

The main components of DSIPTS and its framework, described in Section 2.3, are schematized in Fig. 1 and described in the next section.

The library and the environment are hosted in a public GitLab repository (https://gitlab.fbk.eu/dsip/dsip_dlresearch/timeseries) in which a simple CI/CD pipeline builds the pip package and the documentation (using sphinx) that is served by the GitLab pages service (https://dsip.pages.fbk.eu/dsip_dlresearch/timeseries/). A mirror to GitHub (https://github.com/DSIP-FBK/DSIPTS/) keeps the repository synchronized. The package's README contains some examples and a list of the implemented architectures. At the same time, in the `bash_examples` environment, we explain the training process and the comparison phase with examples using GPUs and a SLURM cluster.

2.2. Software functionalities

A **TimeSeries** object is created from a pandas **DataFrame**. The presence of a column called *time* is the only constraint on the input data set, and it identifies the temporal unit (`integer`, `date`, or `datetime`). The loading procedure finds the smallest frequency and fills the missing values with `NaN`. Samples with `NaN` values will be skipped during the **DataLoader** generation. It is still possible to input missing data before loading the data set. During the definition phase, it is possible to specify the target columns, the categorical variables (supposed to be known in the past and the future), the past numerical variables, and the future known covariates. It is possible to check and remove duplicates (using the time index) and specify if *groups* are present in the data set. With DSIPTS, it is possible to train a single model for different instances of the same source of data (for example, the energy consumption of two apartments): in this case, the index will be the couple (group, time) and the groups will be added as a categorical variable; a single item of the batch is completely referred to a single group avoiding mixing data from different groups. DSIPTS implements an easy interface for loading the most used benchmark time series coming from [3–5].

Moreover, it is possible to specify some standard temporal variables for enriching the initial data set (day of the week, hour, minutes, month) that will be automatically added to the pool of the categorical covariates.

Once the time series is loaded, it is possible to specify the train, validation, and test sets as a percentage of the total number of rows in the DataFrame or use a start and an endpoint for each of the sets. All the numerical variables are normalized, fitting the selected scaler (for example a **StandardScaler**) on the training set, while each categorical variable is encoded with values between 0 and $n_{classes-1}$ using the class **LabelEncoder** from sklearn. A sliding window approach is performed to create the samples (see Fig. 2). The parameters of the splitting procedure are:

- `past_steps` and `future_steps`: number of past and future steps to use in the model
- `shift`: for some models, it is necessary that the first point of the future is the last point of the past. With this parameter, it is possible to control how many past steps to include in the future array.
- `keep_entire_seq_while_shifting`: in the case of a positive shift, this boolean flag indicates if collect the whole sequence of length `shift+future_steps` or just a sequence of length `future_steps`.
- `starting_point`: creating samples aligned on a select column is also possible. For example, if the task is to predict daily sales for the week on Sunday, it is possible to specify that the first output is always on Monday.
- `skip_step`: for a large data set, it is possible to increase the stride of the sliding window to reduce the number of samples.
- `normalize_per_group`: in case of the presence of groups, it specifies whenever the normalization is applied globally or by groups.

A Pytorch **DataLoader** object is generated from the custom **Dataset** relative to the train and validation set. A sample in the **Dataset** (the return of the `__getitem__` function) contains the target variables (`y`), the past (`x_cat_past`) and future (`x_cat_future`) categorical variables, the past (`x_num_past` the only mandatory key) and future (`x_num_future`) numerical covariates and the channel positions of target variables (`idx_target`) inside the past numerical tensor. The data set also contains information about the groups and the time index, which is useful in the inference step for building the resulting dataframe.

A model in DSIPTS is an extension of a base class that implements logging operations at the end of each epoch and contains all the

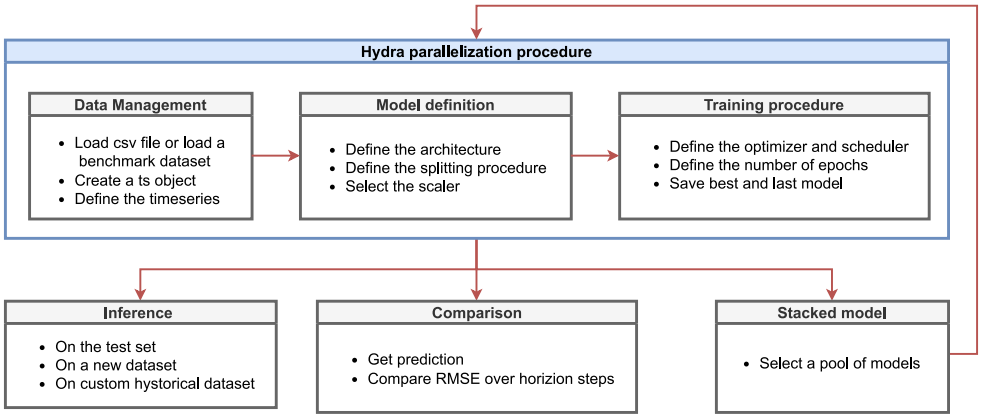


Fig. 1. General framework of the proposed solution.

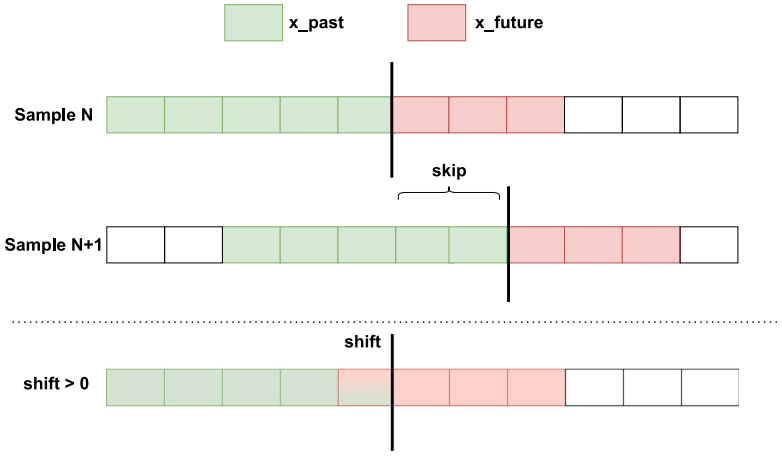


Fig. 2. The first two rows indicate two consecutive samples created using the sliding window (past_steps=5, future_steps=3, skip_step=2). The last row shows how the shift parameter works (in this case the entire future sequence is returned).

implemented loss functions: classical *L1*, *SMAPE* and *MSE* losses but also more complex measures such as *quantile loss* [21], *MDA* Mean directional accuracy [22], *sinkhorn divergence* [23], *dilated loss* [24].

The batch structure defined before ensures the input data always have the same form and dimension: $B \times L \times C$ for the input with B the batch size, L the length, and C the number of channels. $B \times L \times C \times M$ is the output, where M is the number of quantiles (M is equal to 1 if the *quantile loss* is not used). This structure allows the user to add a custom model and a custom loss function easily.

The training procedure is performed by the PyTorch Lightening **Trainer** engine, where callbacks for monitoring the loss function during the train and validation step and saving procedures for the model weights are implemented.

The list of the available DL models is constantly updated, and it currently counts twelve architectures. The Pytorch code is generally taken from the author's repository if it is released with Apache v2.0 or MIT license. Small adjustments are introduced, especially inside the forward function, to match the uniform batch structure described above. In particular: D3VAE [25], LSTF [26], PatchTST [27], Informer [28], Autoformer [3], Crosformer [29], iTransformer [30] and TIDE [31] are taken from Github repositories and the reference are reported inside the code and the README of DISIPTS. The remaining architectures are developed from scratch: Custom Diffusion-based architecture [32] using attention mechanism in the subnets, TFT [11] VQ-VAE [33] based on mingpt [34], Custom Encoder–Decoder architecture mixing Dilated Convolutional layers [35] and RNN layers [36].

The Examples 2 show how to set and train a deep learning model.

Once the training procedure is completed, one can store the **TimeSeries** object and reload it for the inference step. The load function accepts the path to the weights as input, allowing the user to train models in a high-performance environment (GPU-based or cluster) and serve the predictions in a less demanding machine. In DSIPTS, it is possible to load a set (train, validation, or test) using the same parameters as in the training procedure or new defined by the user; eventually, it is possible to provide a new dataframe with (at least) the same columns as the original one. The output of the inference step is a pandas dataframe containing the time index, the groups (if any), the predicted values (in case of quantile loss, multiple columns are generated for a single target variable), the real values (when available), the prediction date and the `lag index`. This last value is crucial for computing the average error along the forecast horizon. Not all the aforementioned tools include this metric; for example, in Darts, the output is a dataframe containing the couple index-value without any specification on the corresponding lag. Example Example 2 shows how to save a trained model, obtain the predictions, and we display the first five rows of the resulting data set.

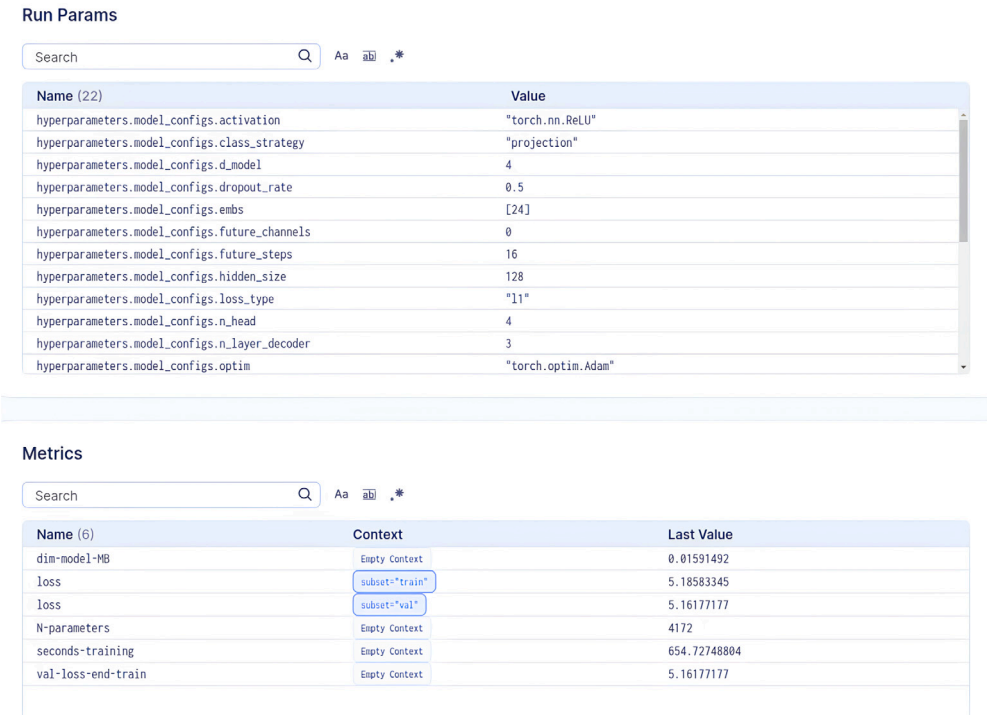


Fig. 3. Screenshot of the aim dashboard reporting the model configuration and some useful metrics.

It is possible to build a stacked generalization model [37] that uses the predictions of other trained models (called base models) as future covariates. In the current formulation, the stacked generalization model is trained on the validation set of the base model to avoid any overfitting problem on the training data set. As an aggregator, it is possible to select one of the implemented architectures that use the future covariates. Once trained, this model is treated as all the others except for a dedicated inference phase composed of a preprocessing step that computes the prediction of the base models.

DSIPTS implements only deep learning architectures trained using the Trainer defined in PyTorch Lightening. This assumption can be limiting when building custom architecture. To extend the usability of DSIPTS, it is possible to use the **Modifier** class to add more complexity to the deep learning architecture. For example, we propose **VVA**: a gpt-based [34,38] deep learning model in which tokens are generated by dividing the signal into small chunks that are clusterized using the k-means algorithm. A Modifier must implement three methods: *fit_transform*, *transform* and *inverse_transform*. The first method is used to modify the train and validation DataLoader; for VVA, a k-means model is fitted on the training data set. The transform method is used in inference modifying the inference DataLoader, and finally, the *inverse_transform* function reverts the model output according to the original output (in VVA, the cluster's centroid is used). A **Modifier** can be used, for example, for building a hybrid forecasting framework [39]. In their work, Di Mauro and colleagues show that mixing statistical methods like the Vector Autoregressive Model (VAR) and classical deep learning architectures (LSTM and GRU) improves the performance obtained by the two approaches separately.

The **Trainer** is paired with an Aim logger [40] that tracks the model parameters, the time required by the training phase, the number of parameters, and the dimension in MB. It also logs the training and validation loss during the training procedure (see Fig. 3). Moreover, every 10% of the total epochs, it also tracks the real and predicted values for the first element in the initial batch of the validation set.

2.3. Environment

The library contains all the logic related to the initialization, saving, and loading procedures, the training step, and the inference phase. Bounded with DSIPTS, we include an environment for training different models on the same data set and comparing the results. Different models on the same data set share many common parameters (paths, splitting extremes, devices, number of workers, etc.) and differ only on the model's settings. With Hydra, it is possible to define a master configuration file initializing all the common settings and create a folder containing one *yaml* file per model specifying its parameters; moreover, it is possible to overwrite any parameter from the command line.

Through Hydra, parallel processes using different protocols can be managed: *joblib* for CPU/GPU architectures and a custom launcher for SLURM clusters [41]. The SLURM launcher automatically allocates the resources needed without breaking policy rules and creates an easily navigable logging tree to control the jobs' status. Example Example 1 shows how to train in parallel three instances of the same model with different values of the parameter d_model.

Example 1. *The command for training in parallel three dlinear models with hidden_size 32, 64, and 128. The logging system writes in the folder multirun/2023-10-20/09-57-45 creating the folders 0, 1, 2 with inside the complete configuration file and the train.log logging file.*

```
$ python train.py --config-name=config architecture=dlinear
  model_configs.hidden_size=32,64,128 -m

Launching jobs, sweep output dir : multirun/2023-10-20/09-57-45
0 : architecture=dlinear model_configs.hidden_size=32
1 : architecture=dlinear model_configs.hidden_size=64
2 : architecture=dlinear model_configs.hidden_size=128
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent
workers.
```

As a last step, Hydra implements a sweeper for Optuna for searching the better configuration among a set of possibilities using an optimized algorithm for searching and pruning strategies.

Fig. 4. Mean percentage error on the test set grouped per lag for the three models trained in Example 1.

Example 2. *Initialization and train of a TimeSeries object using the weather data set.*

```
from dsipts import TimeSeries,read_public_dataset, LinearTS
data, columns = read_public_dataset(dataset = 'weather',
                                    path='/home/TS/data')
ts = TimeSeries(name='weather', stacked=False)
ts.load_signal(data, enrich_cat= ['hour'],target_variables=['y'],
               past_variables=columns if conf.ts.use_covariates
               else [])

past_steps = 16
future_steps = 16
embs =[ts.dataset[c].nunique() for c in ts.cat_var]
config = dict(
    model_configs = dict(
        past_steps = past_steps,
        future_steps = future_steps,
        past_channels = len(ts.num_var),
        future_channels = len(ts.future_variables),
        cat_emb_dim = 8, kernel_size = 3, dropout_rate = 0.25,
        use_bn = True,  optim='torch.optim.Adam',
        activation= 'torch.nn.PReLU',
        sum_emb = True, verbose = False,
        out_channels = len(ts.target_variables),
        hidden_size = 256, kind='dlinear',
        embs = embs, quantiles= [0.1,0.5,0.9] ),
        )
scheduler_config = dict(gamma=0.05,step_size=50)
optim_config = dict(lr = 0.0005,weight_decay=0.01)

model_linear = LinearTS(**config['model_configs'],
                        optim_config = optim_config,
                        scheduler_config =scheduler_config )

ts.set_model(model_linear,config=config ) ##set the model
split_params = {'perc_train':0.7,'perc_valid':0.1,
                'past_steps':past_steps,
                'future_steps':future_steps,
                'shift':0,'starting_point':None,
                'skip_step': 1}
ts.train_model(dirpath=f"/home/TS/model/linear",
               split_params=split_params,
               batch_size=128, num_workers=4,
               max_epochs=50, auto_lr_find=True)

ts.losses.plot()       ## Plot the losses, check overfitting
ts.save("test")        ## Save the TimeSeries object
## load weights from the last step or best step in validation
ts.load(model = LinearTS,"test",load_last=False)
## Inference on the test set and rescaling to the original values
res = ts.inference_on_set(batch_size=200,set='test',rescaling=True)
print(res.sort_values(by=['prediction_time','lag']).head())
```

```
lag       time            y  y_low y_median y_high   prediction_time
1  2020-10-19 22:00:00  438.7  434.3  438.9  441.2  2020-10-19 21:50:00
2  2020-10-19 22:10:00  440.5  432.8  438.0  445.8  2020-10-19 21:50:00
3  2020-10-19 22:20:00  435.2  432.3  440.1  446.2  2020-10-19 21:50:00
4  2020-10-19 22:30:00  437.2  432.5  440.8  447.5  2020-10-19 21:50:00
5  2020-10-19 22:40:00  440.8  433.8  440.4  450.4  2020-10-19 21:50:00
```

Example 3. *Comparison of the three models trained in Example 1. The mean average percentage errors per lag on the tes are displayed in Fig. 4.*

```
$ python compare.py  --config-name=compare
```

From Python:

```
import plotly.express as px
import pandas as pd
import numpy as np
tot_pred = pd.read_csv('/home/TS/model/linear/csv/tot_predictions.csv')
error = tot_pred.groupby(['lag','model']).apply(lambda x:
                                100*np.nanmean(np.abs(x.y-x.y_pred)/x.y)
                                ).reset_index()
error.rename(columns={0:'error'},inplace=True)
fig = px.line(error, x='lag', y='error', color='model',
              height=600,width=1000)
fig.update_layout(legend=dict(x=0.65,y=.1),font=dict(size=16))
fig.write_image('error.pdf')
```

3. Illustrative examples

Ten architectures with standard parameters have been tested on two benchmark datasets from [3] (namely weather and etth) on a PC with 32 GB of RAM, an Intel i7-6700HQ CPU @ 2.60 GHz processor with GPU NVIDIA RTX A4000. All the models have been trained for 100 epochs using the command:

```
python train.py –config-dir=config_SX –config-name=config_xps dataset.dataset=weather,etth1 -m
```

and some statistics (number of parameters, training time, and final validation loss) are reported in Table 1. The models have been trained using 60% of the data (30k for weather, 10k for etth) with `past_steps` and `future_steps` equal to 64.

4. Impact

With DSIPTS, it is possible to efficiently train different instances of the same DL architecture or different architectures on the same training set and quickly compare the performances on the selected test set. The automatized batch-generation process allows the user to implement a custom model focusing only on the architecture definition. The inference procedure is designed to generate a prediction in a production environment since it requires feeding the trained model with the most recent raw dataset. During this phase, all the scalers are applied to prepare the data loader based on the parameters provided during the training phase. The variety of loss functions implemented can help

Table 1
Performance of 10 architectures on the weather and etth dataset. Each architecture has been trained for 100 epochs without any fine tuning of the parameters with two parallel jobs (except for * that runs as single job).

	Weather			Etth		
	#par	time	val loss	#par	time	val loss
autoformer	286k	10 h 35 m	1.01	291k	3 h	0.75
crossformer	850k	1 h 42 m	0.56	810k	29 m	0.38
dilated conv	66k	14 m	0.37	56k	4 m	0.48
LTSF	10k	4 m	1.01	10k	2 m	0.77
RNN	730k	12 m	0.58	730k	4 m	0.52
informer	311k	1 h 30 m	0.63	311k	31 m	0.42
Itransformer	270k	19 m	0.54	270k	5 m	0.21
patchTST	1.75M	21 h 31 m	0.72	1.75M	10 h 33 m	0.2
TFT	587k	2 h 19 m	0.67	585k	26 m	0.31
TIDE	14.2M	1 h 5 m*	0.6	14.2M	1 h 30 m	0.21

the user select the most suitable based on the problem, for example, penalizing more a miss-predicted output shape using the dilated loss. Moreover, it is possible to build a single model for different identities sharing the same data structure using the groups. Finally, using the modifiers, it is possible to generate more complex models combining standard machine learning techniques, such as clustering, before training one of the DL architectures. All these features allow the researcher to develop new solutions focusing only on the model definition and forward/inference step and quickly deploy and maintain forecasting solutions based on DSIPTS. DSIPTS is currently used inside the Horizon project INCUBE (grant agreement n.101069610) for training predictive models for energy demand and consumption in smart buildings.

5. Conclusions

We presented DSIPTS, a new compact and ready-to-use library for time series forecasting, to test new architectures and compare them with standard models using benchmark data sets. In DSIPTS, the batch structure contains all the information in a standardized way to develop new solutions rapidly. DSIPTS is coupled with a framework based on Hydra, allowing parallel training sessions. The main difference between similar solutions relies on the expandability, the transparency of the inference phase, the importance of future and categorical variables, and the possibility of using more complex loss functions to train the models.

CRediT authorship contribution statement

Andrea Gobbi: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Andrea Martinelli:** Software. **Marco Cristoforetti:** Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data mentioned in the paper can be downloaded freely from the web.

References

[1] Falcon W, The PyTorch Lightning team. PyTorch lightning. 2019, http://dx.doi.org/10.5281/zenodo.3828935, URL https://github.com/Lightning-AI/lightning.
[2] MetaResearch. Hydra. 2023, https://github.com/facebookresearch/hydra.
[3] Wu H, Xu J, Wang J, Long M. Autoformer: Decomposition transformers with Auto-Correlation for long-term series forecasting. 2021,
[4] Godahewa R, Bergmeir C, Webb GI, Hyndman RJ, Montero-Manso P. Monash time series forecasting archive. In: Neural information processing systems track on datasets and benchmarks. 2021.
[5] Ughi R, Lomurno E, Matteucci M. Two steps forward and one behind: Rethinking time series forecasting with deep learning. 2023, http://dx.doi.org/10.48550/ARXIV.2304.04553, URL https://arxiv.org/abs/2304.04553.
[6] Zhang C. Time series forecasting and deep learning. 2023, https://github.com/DaoSword/Time-Series-Forecasting-and-Deep-Learning#Libraries.
[7] Herzen J, Lassig F, Piazzetta SG, Neuer T, Tafti L, Raille G, et al. Darts: User-friendly modern machine learning for time series. J Mach Learn Res 2022;23(124):1–6, URL http://jmlr.org/papers/v23/21-1177.html.
[8] Liaw R, Liang E, Nishihara R, Moritz P, Gonzalez JE, Stoica I. Tune: A research platform for distributed model selection and training. 2018, arXiv preprint arXiv:1807.05118.
[9] Beitner J, Contributors. PyTorch forecasting. 2023, https://pytorch-forecasting.readthedocs.io/en/stable/.
[10] Akiba T, Sano S, Yanase T, Ohta T, Koyama M. Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery and data mining. 2019.
[11] Lim B, Arık SÖ, Loeff N, Pfister T. Temporal fusion transformers for interpretable multi-horizon time series forecasting. Int J Forecast 2021;37(4):1748–64. http://dx.doi.org/10.1016/j.ijforecast.2021.03.012.
[12] Sktime. Sktime. 2023, https://github.com/sktime/sktime.
[13] MetaResearch. Kats. 2023, https://github.com/facebookresearch/Kats.
[14] MetaResearch. Prophet. 2023, https://facebook.github.io/prophet/.
[15] Catlin C, Contributors. Sktime. 2023, https://github.com/winedarksea/autots.
[16] Alexandrov A, Benidis K, Bohlke-Schneider M, Flunkert V, Gasthaus J, Januschowski T, et al. GluonTS: Probabilistic and neural time series modeling in Python. J Mach Learn Res 2020;21(116):1–6, URL http://jmlr.org/papers/v21/19-820.html.
[17] Chen T, Li M, Li Y, Lin M, Wang N, Wang M, et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015.
[18] The pandas development team. pandas-dev/pandas: Pandas. URL https://github.com/pandas-dev/pandas.
[19] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62. http://dx.doi.org/10.1038/s41586-020-2649-2.
[20] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[21] Koenker R, Bassett G. Regression quantiles. Econometrica 1978;46(1):33. http://dx.doi.org/10.2307/1913643.
[22] Pesaran MH, Timmermann A. How costly is it to ignore breaks when forecasting the direction of a time series? Int J Forecast 2004;20(3):411–25. http://dx.doi.org/10.1016/S0169-2070(03)00068-2, URL https://www.sciencedirect.com/science/article/pii/S0169207003000682.
[23] Cuturi M. Sinkhorn distances: Lightspeed computation of optimal transport. Adv Neural Inf Process Syst 2013;26. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/af21d0c97db2e27e13572cbf59eb343d-Paper.pdf.
[24] Sakoe H, Chiba S. Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans Acoust Speech Signal Process 1978;26(1):43–9. http://dx.doi.org/10.1109/TASSP.1978.1163055.
[25] Li Y, Lu X, Wang Y, Dou D. Generative time series forecasting with diffusion, denoise, and disentanglement. 2022, URL https://openreview.net/forum?id=rG0jm74xtx,
[26] Association for Artificial Intelligence 2023, Chen M, Xu Q, Zeng A, Zhang L. Are transformers effective for time series forecasting? Underline Science Inc; 2023, http://dx.doi.org/10.48448/ZKAZ-JK77, URL https://underline.io/lecture/67908-are-transformers-effective-for-time-series-forecastingquestion.
[27] Nie Y, H. Nguyen N, Sinthong P, Kalagnanam J. A time series is worth 64 words: Long-term forecasting with transformers. In: International conference on learning representations. 2023.
[28] Zhou H, Zhang S, Peng J, Zhang S, Li J, Xiong H, et al. Informer: Beyond efficient transformer for long sequence time-series forecasting. In: Proceedings of the AAAI conference on artificial intelligence, vol. 35, (no. 12):2021, p. 11106–15. http://dx.doi.org/10.1609/aaai.v35i12.17325, URL https://ojs.aaai.org/index.php/AAAI/article/view/17325.
[29] Zhang Y, Yan J. Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting. In: The eleventh international conference on learning representations. 2023, URL https://openreview.net/forum?id=vSVLM2j9eie.
[30] Liu Y, Hu T, Zhang H, Wu H, Wang S, Ma L, et al. Itransformer: Inverted transformers are effective for time series forecasting. 2023, http://dx.doi.org/10.48550/ARXIV.2310.06625, URL https://arxiv.org/abs/2310.06625.

[31] Das A, Kong W, Leach A, Mathur S, Sen R, Yu R. Long-term forecasting with TiDE: Time-series dense encoder. 2023, http://dx.doi.org/10.48550/ARXIV.2304.08424, URL https://arxiv.org/abs/2304.08424.

[32] Nichol AQ, Dhariwal P. Improved denoising diffusion probabilistic models. In: Meila M, Zhang T, editors. Proceedings of the 38th international conference on machine learning. Proceedings of machine learning research, 139, PMLR; 2021, p. 8162–71, URL https://proceedings.mlr.press/v139/nichol21a.html.

[33] Oord Avd, Vinyals O, Kavukcuoglu K. Neural discrete representation learning. 2017, http://dx.doi.org/10.48550/ARXIV.1711.00937, URL https://arxiv.org/abs/1711.00937.

[34] Karpathy A. Mingpt. 2023, https://github.com/karpathy/minGPT.

[35] Yu F, Koltun V. Multi-scale context aggregation by dilated convolutions. 2015, http://dx.doi.org/10.48550/ARXIV.1511.07122, URL https://arxiv.org/abs/1511.07122.

[36] Cho K, van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In: Proceedings of the 2014 conference on empirical methods in natural language processing. Doha, Qatar: Association for Computational Linguistics; 2014, p. 1724–34. http://dx.doi.org/10.3115/v1/D14-1179, URL https://aclanthology.org/D14-1179.

[37] Wolpert DH. Stacked generalization. Neural Netw 1992;5(2):241–59. http://dx.doi.org/10.1016/S0893-6080(05)80023-1, URL https://www.sciencedirect.com/science/article/pii/S0893608005800231.

[38] Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I. Language models are unsupervised multitask learners. 2019, URL https://api.semanticscholar.org/CorpusID:160025533.

[39] Di Mauro M, Galatro G, Postiglione F, Song W, Liotta A. Hybrid learning strategies for multivariate time series forecasting of network quality metrics. Comput Netw 2024;243:110286. http://dx.doi.org/10.1016/j.comnet.2024.110286.

[40] Arakelyan G, Soghomonyan G, The Aim team. Aim. 2020, http://dx.doi.org/10.5281/zenodo.6536395, URL https://github.com/aimhubio/aim.

[41] SchedMD. Slurm. 2023, https://slurm.schedmd.com/overview.html.