Flow Forecast

Release 0.0.1

Isaac Godfried

May 19, 2021

GENERAL:

Flow Forecast is a deep learning for time series forecasting framework written in PyTorch. Flow Forecast makes it easy to train PyTorch Forecast models on a wide variety of datasets.

CHAPTER

ONE

UTILITIES

flood_forecast.utils.**numpy_to_tvar**(*x*)
Converts a numpy array into a PyTorch Tensor

Parameters **x** (*[type]*) – A numpy array you want to convert

Returns [description]

Return type torch.Variable

flood_forecast.utils.**flatten_list_function**(*input_list: List*)

flood_forecast.utils.**make_criterion_functions**(*crit_list*) → List
crit_list should be either dict or list

CHAPTER

TWO

MODEL EVALUATION

flood_forecast.evaluator.**stream_baseline**(*river_flow_df: pandas.core.frame.DataFrame*, *forecast_column: str*, *hours_forecast=336) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'float'>*)

Function to compute the baseline MSE by using the mean value from the train data.

flood_forecast.evaluator.**plot_r2**(*river_flow_preds: pandas.core.frame.DataFrame*) → float

We assume at this point river_flow_preds already has a predicted_baseline and a predicted_model column

flood_forecast.evaluator.**get_model_r2_score**(*river_flow_df: pandas.core.frame.DataFrame*, *model_evaluate_function: Callable*, *forecast_column: str*, *hours_forecast=336*)

model_evaluate_function should call any necessary preprocessing

flood_forecast.evaluator.**get_r2_value**(*model_mse*, *baseline_mse*)

flood_forecast.evaluator.**get_value**(*the_path: str*) → None

flood_forecast.evaluator.**evaluate_model**(*model: Type[*flood_forecast.time_model.TimeSeriesModel*]*, *model_type: str*, *target_col: List[str]*, *evaluation_metrics: List*, *inference_params: Dict*, *eval_log: Dict*) → Tuple[Dict, pandas.core.frame.DataFrame, int, pandas.core.frame.DataFrame]

A function to evaluate a model. Called automatically at end of training. Can be imported for continuing to evaluate a model in other places as well.

```
from flood_forecast.evaluator import evaluate_model
forecast_model = PyTorchForecast(config_file)
e_log, df_train_test, f_idx, df_preds = evaluate_model(forecast_model, "PyTorch",␣
↪["cfs"], ["MSE", "MAPE"], {})
print(e_log) # {"MSE":0.2, "MAPE":0.1}
print(df_train_test) #
...
```

“”

flood_forecast.evaluator.**infer_on_torch_model**(*model, test_csv_path: Optional[str] = None, datetime_start: datetime.datetime = datetime.datetime(2018, 9, 22, 0, 0), hours_to_forecast: int = 336, decoder_params=None, dataset_params: Dict = {}, num_prediction_samples: Optional[int] = None, probabilistic: bool = False, criterion_params: Optional[Dict] = None) -> (<class 'pandas.core.frame.DataFrame'>, <class 'torch.Tensor'>, <class 'int'>, <class 'int'>, <class 'flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader'>, typing.List[pandas.core.frame.DataFrame]*)

Function to handle both test evaluation and inference on a test data-frame. :return:

df: df including training and test data end_tensor: the final tensor after the model has finished predictions history_length: num rows to use in training forecast_start_idx: row index to start forecasting test_data: CSVTestLoader instance df_prediction_samples: has same index as df, and num cols equal to num_prediction_samples or no columns if num_prediction_samples is None

Return type tuple()

flood_forecast.evaluator.**handle_ci_multi**(*prediction_samples: torch.Tensor, csv_test_loader: flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader, multi_params: int, df_pred, decoder_param: bool, history_length: int, num_samples: int*) → List[pandas.core.frame.DataFrame]

[summary]

Parameters

- **prediction_samples** (*torch.Tensor*) – [description]
- **csv_test_loader** (CSVTestLoader) – [description]
- **multi_params** (*int*) – [description]
- **df_pred** (*[type]*) – [description]
- **decoder_param** (*bool*) – [description]
- **history_length** (*int*) – [description]
- **num_samples** (*int*) – [description]

Raises

- **ValueError** – [description]
- **ValueError** – [description]

Returns Returns an array with different CI predictions

Return type List[pd.DataFrame]

flood_forecast.evaluator.**generate_predictions**(*model: Type[flood_forecast.time_model.TimeSeriesModel]*, *df: pandas.core.frame.DataFrame*, *test_data: flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader*, *history: torch.Tensor*, *device: torch.device*, *forecast_start_idx: int*, *forecast_length: int*, *hours_to_forecast: int*, *decoder_params: Dict*, *targs=False*, *multi_params: int = 1*) → torch.Tensor

A function to generate the actual model prediction

Parameters

- **model** (*Type[TimeSeriesModel]*) – A PyTorchForecast
- **df** (*pd.DataFrame*) – The main dataframe containing data
- **test_data** (CSVTestLoader) – The test data loader
- **history** (*torch.Tensor*) – The forecast historical data
- **device** (*torch.device*) – The device usually cpu or cuda
- **forecast_start_idx** (*int*) – The index you want the forecast to begin
- **forecast_length** (*int*) – The length of the forecast the model outputs per time step
- **hours_to_forecast** (*int*) – The number of time_steps to forecast in future
- **decoder_params** (*Dict*) – The parameters the decoder function takes.
- **multi_params** (*int, optional*) – n_targets, defaults to 1

Returns The forecasted tensor

Return type torch.Tensor

flood_forecast.evaluator.**generate_predictions_non_decoded**(*model: Type[flood_forecast.time_model.TimeSeriesMod*, *df: pandas.core.frame.DataFrame*, *test_data: flood_forecast.preprocessing.pytorch_loaders.CS*, *history_dim: torch.Tensor*, *forecast_length: int*, *hours_to_forecast: int*) → torch.Tensor

Generates predictions for the models that do not use a decoder

Parameters

- **model** (*Type[TimeSeriesModel]*) – A PyTorchForecast
- **df** (*pd.DataFrame*) – [description]
- **test_data** (CSVTestLoader) – [description]
- **history_dim** (*torch.Tensor*) – [description]
- **forecast_length** (*int*) – [description]
- **hours_to_forecast** (*int*) – [description]

Returns [description]

Return type torch.Tensor

flood_forecast.evaluator.**generate_decoded_predictions**(*model: Type[flood_forecast.time_model.TimeSeriesModel], test_data: flood_forecast.preprocessing.pytorch_loaders.CSVTes forecast_start_idx: int, device: torch.device, history_dim: torch.Tensor, hours_to_forecast: int, decoder_params: Dict, multi_targets: int = 1, targs: Union[bool, torch.Tensor] = False*) → torch.Tensor

flood_forecast.evaluator.**generate_prediction_samples**(*model: Type[flood_forecast.time_model.TimeSeriesModel], df: pandas.core.frame.DataFrame, test_data: flood_forecast.preprocessing.pytorch_loaders.CSVTestL history: torch.Tensor, device: torch.device, forecast_start_idx: int, forecast_length: int, hours_to_forecast: int, decoder_params: Dict, num_prediction_samples: int, multi_params=1, targs=False*) → numpy.ndarray

ss

CHAPTER

THREE

LONG TRAIN

flood_forecast.long_train.**split_on_letter**(*s: str*) → List

flood_forecast.long_train.**loop_through**(*data_dir: str*, *interrmittent_gcs: bool = False*, *use_transfer: bool = True*, *start_index: int = 0*, *end_index: int = 25*) → None

Function that makes and executes a set of config files This is since we have over 9k files and.

flood_forecast.long_train.**make_config_file**(*flow_file_path: str*, *gage_id: str*, *station_id: str*, *weight_path: Optional[str] = None*)

flood_forecast.long_train.**main**()

CHAPTER

FOUR

MODEL DICTIONARIES

CHAPTER

FIVE

PRE DICTIONARIES

CHAPTER

SIX

PYTORCH TRAINING

flood_forecast.pytorch_training.**handle_meta_data**(*model:* flood_forecast.time_model.PyTorchForecast)

A function to init models with meta-data s :param model: A PyTorchForecast model with meta_data parameter block in config file. :type model: PyTorchForecast :return: Returns a tuple of the initial meta-representationf :rtype: tuple(PyTorchForecast, torch.Tensor, torch.nn)

flood_forecast.pytorch_training.**train_transformer_style**(*model:* flood_forecast.time_model.PyTorchForecast, *training_params: Dict*, *takes_target=False*, *forward_params: Dict = {}*, *model_filepath: str = 'model_save'*) → None

Function to train any PyTorchForecast model

Parameters

- **model** (*PyTorchForecast*) – A properly wrapped PyTorchForecast model
- **training_params** (*Dict*) – A dictionary of the necessary parameters for training.
- **takes_target** (*bool, optional*) – A parameter to determine whether a model requires the target, defaults to False
- **forward_params** (*Dict, optional*) – [description], defaults to {}
- **model_filepath** (*str, optional*) – The file path to load modeel weights from, defaults to "model_save"

Raises **ValueError** – Has an error

flood_forecast.pytorch_training.**get_meta_representation**(*column_id: str*, *uuid: str*, *meta_model:* flood_forecast.time_model.PyTorchForecast) → torch.Tensor

flood_forecast.pytorch_training.**handle_scaling**(*validation_dataset*, *src*, *output: torch.Tensor*, *labels*, *probabilistic*, *m*, *output_std*)

flood_forecast.pytorch_training.**compute_loss**(*labels*, *output*, *src*, *criterion*, *validation_dataset*, *probabilistic=None*, *output_std=None*, *m=1*)

Function for computing the loss

Parameters

- **labels** (*torch.Tensor*) – The real values for the target. Shape can be variable but should follow (batch_size, time)

- **output** (*torch.Tensor*) – The output of the model
- **src** (*torch.Tensor*) – The source values (only really needed for the MASELoss function)
- **criterion** (*[type]*) – [description]
- **validation_dataset** (*torch.utils.data.dataset*) – Only passed when unscaling of data is needed.
- **probabilistic** (*[type], optional*) – Whether the model is a probabalistic returns a distribution, defaults to None
- **output_std** (*[type], optional*) – The standard distribution, defaults to None
- **m** (*int, optional*) – The number of targets defaults to 1

Returns Returns the computed loss

Return type float

flood_forecast.pytorch_training.**torch_single_train**(*model: flood_forecast.time_model.PyTorchForecast, opt: torch.optim.optimizer.Optimizer, criterion: Type[torch.nn.modules.loss._Loss], data_loader: torch.utils.data.dataloader.DataLoader, takes_target: bool, meta_data_model: flood_forecast.time_model.PyTorchForecast, meta_data_model_representation: torch.Tensor, meta_loss=None, multi_targets=1, forward_params: Dict = {}*) → float

flood_forecast.pytorch_training.**multi_step_forecasts_append**(*self*)

Function to handle forecasts that span multiple time steps

flood_forecast.pytorch_training.**compute_validation**(*validation_loader: torch.utils.data.dataloader.DataLoader, model, epoch: int, sequence_size: int, criterion: Type[torch.nn.modules.loss._Loss], device: torch.device, decoder_structure=False, meta_data_model=None, use_wandb: bool = False, meta_model=None, multi_targets=1, val_or_test='validation_loss', probabilistic=False*) → float

Function to compute the validation loss metrics

Parameters

- **validation_loader** (*DataLoader*) – The data-loader of either validation or test-data
- **model** (*[type]*) – model

- **epoch** (*int*) – The epoch where the validation/test loss is being computed.
- **sequence_size** (*int*) – [description]
- **criterion** (*Type[torch.nn.modules.loss._Loss]*) – [description]
- **device** (*torch.device*) – The device
- **decoder_structure** (*bool, optional*) – Whether the model should use sequential decoding, defaults to False
- **meta_data_model** (PyTorchForecast, *optional*) – The model to handle the meta-data, defaults to None
- **use_wandb** (*bool, optional*) – Whether Weights and Biases is in use, defaults to False
- **meta_model** (*bool, optional*) – Whether the model leverages meta-data, defaults to None
- **multi_targets** (*int, optional*) – Whether the model, defaults to 1
- **val_or_test** (*str, optional*) – Whether validation or test loss is computed, defaults to "validation_loss"
- **probabilistic** (*bool, optional*) – Whether the model is probablistic, defaults to False

Returns The loss of the first metric in the list.

Return type float

CHAPTER

SEVEN

TIME MODEL

class flood_forecast.time_model.**TimeSeriesModel**(*model_base: str*, *training_data: str*, *validation_data: str*, *test_data: str*, *params: Dict*)

An abstract class used to handle different configurations of models + hyperparams for training, test, and predict functions. This class assumes that data is already split into test train and validation at this point.

__init__(*model_base: str*, *training_data: str*, *validation_data: str*, *test_data: str*, *params: Dict*)
Initialize self. See help(type(self)) for accurate signature.

abstract load_model(*model_base: str*, *model_params: Dict*, *weight_path=None*) → object
This function should load and return the model this will vary based on the underlying framework used

abstract make_data_load(*data_path*, *params: Dict*, *loader_type: str*) → object
Intializes a data loader based on the provided data_path. This may be as simple as a pandas dataframe or as complex as a custom PyTorch data loader.

abstract save_model(*output_path: str*)
Saves a model to a specific path along with a configuration report of the parameters and data info.

upload_gcs(*save_path: str*, *name: str*, *file_type: str*, *epoch=0*, *bucket_name=None*)
Function to upload model checkpoints to GCS

wandb_init()

class flood_forecast.time_model.**PyTorchForecast**(*model_base: str*, *training_data*, *validation_data*, *test_data*, *params_dict: Dict*)

__init__(*model_base: str*, *training_data*, *validation_data*, *test_data*, *params_dict: Dict*)
Initialize self. See help(type(self)) for accurate signature.

load_model(*model_base: str*, *model_params: Dict*, *weight_path: Optional[str] = None*, *strict=True*)
This function should load and return the model this will vary based on the underlying framework used

save_model(*final_path: str*, *epoch: int*) → None
Function to save a model to a given file path

make_data_load(*data_path: str*, *dataset_params: Dict*, *loader_type: str*, *the_class='default'*)
Intializes a data loader based on the provided data_path. This may be as simple as a pandas dataframe or as complex as a custom PyTorch data loader.

flood_forecast.time_model.**scaling_function**(*start_end_params*, *dataset_params*)

CHAPTER

EIGHT

TRAINER

flood_forecast.trainer.**train_function**(*model_type: str*, *params: Dict*)

Function to train a Model(TimeSeriesModel) or da_rnn. Will return the trained model :param model_type: Type of the model. In almost all cases this will be 'PyTorch' :type model_type: str :param params: Dictionary containing all the parameters needed to run the model :type Dict:

flood_forecast.trainer.**main**()

Main function which is called from the command line. Entrypoint for training all ML models.

CHAPTER

NINE

INFERENCE

This API makes it easy to run inference on trained PyTorchForecast modules. To use this code you need three main files: your model's configuration file, a CSV containing your data, and a path to your model weights.

Listing 1: example initialization

```
import json
from datetime import datetime
from flood_forecast.deployment.inference import InferenceMode
new_water_data_path = "gs://predict_cfs/day_addition/01046000KGNR_flow.csv"
weight_path = "gs://predict_cfs/experiments/10_December_202009_34PM_model.pth"
with open("config.json") as y:
  config_test = json.load(y)
infer_model = InferenceMode(336, 30, config_test, new_water_data_path, weight_path,
↪"river")
```

Listing 2: example plotting

```
```

class flood_forecast.deployment.inference.**InferenceMode**(*forecast_steps: int*, *num_prediction_samples: int*, *model_params*, *csv_path: str*, *weight_path*, *wandb_proj: Optional[str] = None*, *torch_script=False*)

__init__(*forecast_steps: int*, *num_prediction_samples: int*, *model_params*, *csv_path: str*, *weight_path*, *wandb_proj: Optional[str] = None*, *torch_script=False*)

Class to handle inference for models,

Parameters

- **forecasts_steps** – Number of time-steps to forecast (doesn't have to be hours)
- **num_prediction_samples** (*int*) – Number of prediction samples
- **model_params** (*Dict*) – A dictionary of model parameters (ideally this should come from saved JSON config file)
- **csv_path** (*str*) – Path to the CSV test file you want to be used for inference. Evem of you aren't using
- **weight_path** (*str*) – Path to the model weights

- **wandb_proj** (*str, optionals*) – The name of the WB project leave blank if you don't want to log to Wandb, defaults to None

infer_now(*some_date: datetime.datetime*, *csv_path=None*, *save_buck=None*, *save_name=None*, *use_torch_script=False*)

Performs inference on a CSV file at a specified datatime

Parameters

- **some_date** – The date you want inference to begin on.
- **csv_path** (*str, optional*) – A path to a CSV you want to perform inference on, defaults to None
- **save_buck** (*str, optional*) – The GCP bucket where you want to save predictions, defaults to None
- **save_name** (*str, optional*) – The name of the file to save the Pandas data-frame to GCP as, defaults to None
- **use_torch_script** – Optional parameter which allows you to use a saved torch script version of your model.

Returns Returns a tuple consisting of the Pandas dataframe with predictions + history,

the prediction tensor, a tensor of the historical values, the forecast start index, the test loader, and the a dataframe of the prediction samples (e.g. the confidence interval preds) :rtype: tuple(pd.DataFrame, torch.Tensor, int, CSVTestLoader, pd.DataFrame)

make_plots(*date: datetime.datetime*, *csv_path: Optional[str] = None*, *csv_bucket: Optional[str] = None*, *save_name=None*, *wandb_plot_id=None*)

Function to create plots in inference mode.

Parameters

- **date** (*datetime*) – The datetime to start inference
- **csv_path** (*str, optional*) – The path to the CSV file you want to use for inference, defaults to None
- **csv_bucket** (*str, optional*) – [description], defaults to None
- **save_name** (*[type], optional*) – [description], defaults to None
- **wandb_plot_id** (*[type], optional*) – [description], defaults to None

Returns [description]

Return type tuple(torch.Tensor, torch.Tensor, *CSVTestLoader*, matplotlib.pyplot.plot)

flood_forecast.deployment.inference.**convert_to_torch_script**(*model: flood_forecast.time_model.PyTorchForecast*, *save_path: str*) → *flood_forecast.time_model.PyTorchForecast*

Function to convert PyTorch model to torch script and save

Parameters

- **model** (PyTorchForecast) – The PyTorchForecast model you wish to convert
- **save_path** (*str*) – File name to save the TorchScript model under.

Returns Returns the model with an added .script_model attribute

Return type *PyTorchForecast*

flood_forecast.deployment.inference.**convert_to_onnx**()

flood_forecast.deployment.inference.**load_model**(*model_params_dict*, *file_path: str*, *weight_path: str*) → *flood_forecast.time_model.PyTorchForecast*

Function to load a PyTorchForecast model from an existing config file.

Parameters

- **model_params_dict** (*Dict*) – Dictionary of model parameters
- **file_path** (*str*) – [description]
- **weight_path** (*str*) – [description]

Returns [description]

Return type *PyTorchForecast*

CHAPTER

TEN

INTERPOLATE PREPROCESSING

This module allows easy pre-processing of data.

flood_forecast.preprocessing.interpolate_preprocess.**fix_timezones**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Basic function to fix initil data bug related to NaN values in non-eastern-time zones due to UTC conversion.

flood_forecast.preprocessing.interpolate_preprocess.**interpolate_missing_values**(*df: pandas.core.frame.D* → pandas.core.frame.D

Function to fill missing values with nearest value. Should be run only after splitting on the NaN chunks.

flood_forecast.preprocessing.interpolate_preprocess.**forward_back_generic**(*df: pandas.core.frame.DataFram relevant_columns: List*) → pandas.core.frame.DataFrame

Function to fill missing values with nearest value (forward first)

flood_forecast.preprocessing.interpolate_preprocess.**back_forward_generic**(*df: pandas.core.frame.DataFram relevant_columns: List[str]*) → pandas.core.frame.DataFrame

Function to fill missing values with nearest values (backward first)

CHAPTER

ELEVEN

BUILD DATASET

flood_forecast.preprocessing.buil_dataset.**build_weather_csv**(*json_full_path*, *asos_base_url*, *base_url_2*, *econet_data*, *visited_gages_path*, *start=0*, *end_index=100*)

flood_forecast.preprocessing.buil_dataset.**join_data**(*weather_csv*, *meta_json_file*, *flow_csv*)

flood_forecast.preprocessing.buil_dataset.**create_visited**()

flood_forecast.preprocessing.buil_dataset.**get_eco_netset**(*directory_path: str*) → set

Econet data was supplied to us by the NC State climate office. They gave us a directory of CSV files in following format *LastName_First_station_id_Hourly.txt* This code simply constructs a set of stations based on what is in the folder.

flood_forecast.preprocessing.buil_dataset.**combine_data**(*flow_df: pandas.core.frame.DataFrame*, *precip_df: pandas.core.frame.DataFrame*)

flood_forecast.preprocessing.buil_dataset.**create_usgs**(*meta_data_dir: str*, *precip_path: str*, *start: int*, *end: int*)

flood_forecast.preprocessing.buil_dataset.**get_data**(*file_path: str*, *gcp_service_key: Optional[str] = None*) → str

Extract bucket name and storage object name from file_path Args:

file_path (str): [description]

Example, file_path = "gs://task_ts_data/2020-08-17/Afghanistan____.csv" bucket_name = "task_ts_data" object_name = "2020-08-17/Afghanistan____.csv" loal_temp_filepath = "//data/2020-08-17/Afghanistan____.csv"

Returns: str: local file name

CHAPTER

TWELVE

CLOSEST STATION

flood_forecast.preprocessing.closest_station.**get_closest_gage**(*gage_df: pandas.core.frame.DataFrame*, *station_df: pandas.core.frame.DataFrame*, *path_dir: str*, *start_row: int*, *end_row: int*)

flood_forecast.preprocessing.closest_station.**haversine**(*lon1*, *lat1*, *lon2*, *lat2*)

Calculate the great circle distance between two points on the earth (specified in decimal degrees)

flood_forecast.preprocessing.closest_station.**get_weather_data**(*file_path: str*, *econet_gages: Set*, *base_url: str*)

Function that retrieves if station has weather data for a specific gage either from ASOS or ECONet

flood_forecast.preprocessing.closest_station.**format_dt**(*date_time_str: str*) → datetime.datetime

flood_forecast.preprocessing.closest_station.**convert_temp**(*temparature: str*) → float

Note here temp could be a number or 'M' which stands for missing. We use 50 at the moment to fill missing values.

flood_forecast.preprocessing.closest_station.**process_asos_data**(*file_path: str*, *base_url: str*) → Dict

Function that saves the ASOS data to CSV uses output of get weather data.

flood_forecast.preprocessing.closest_station.**process_asos_csv**(*path: str*)

CHAPTER

THIRTEEN

DATA CONVERTER

This module holds functions to convert data effectively.

A set of function aimed at making it easy to convert other time series datasets to our format for transfer learning purposes

flood_forecast.preprocessing.data_converter.**make_column_names**(*df*)

CHAPTER

FOURTEEN

PREPROCESS DA RNN

flood_forecast.preprocessing.preprocess_da_rnn.**format_data**(*dat*, *targ_column: List[str]*) → flood_forecast.da_rnn.custom_types.TrainData

flood_forecast.preprocessing.preprocess_da_rnn.**make_data**(*csv_path: str*, *target_col: List[str]*, *test_length: int*, *relevant_cols=['cfs', 'temp', 'precip']*) → flood_forecast.da_rnn.custom_types.TrainData

Returns full preprocessed data. Does not split train/test that must be done later.

CHAPTER

FIFTEEN

PREPROCESS METADATA

flood_forecast.preprocessing.preprocess_metadata.**make_gage_data_csv**(*file_path: str*)

returns df

CHAPTER

SIXTEEN

PROCESS USGS

flood_forecast.preprocessing.process_usgs.**make_usgs_data**(*start_date: datetime.datetime*, *end_date: datetime.datetime*, *site_number: str*) → pandas.core.frame.DataFrame

flood_forecast.preprocessing.process_usgs.**process_response_text**(*file_name: str*) → Tuple[str, Dict]

flood_forecast.preprocessing.process_usgs.**df_label**(*usgs_text: str*) → str

flood_forecast.preprocessing.process_usgs.**create_csv**(*file_path: str*, *params_names: dict*, *site_number: str*)

Function that creates the final version of the CSV files

flood_forecast.preprocessing.process_usgs.**get_timezone_map**()

flood_forecast.preprocessing.process_usgs.**process_intermediate_csv**(*df: pandas.core.frame.DataFrame) -> (<class 'pandas.core.frame.DataFrame'>, <class 'int'>, <class 'int'>, <class 'int'>)*

CHAPTER

SEVENTEEN

PYTORCH LOADERS

class flood_forecast.preprocessing.pytorch_loaders.**CSVDataLoader**(*file_path: str*, *forecast_history: int*, *forecast_length: int*, *target_col: List*, *relevant_cols: List*, *scaling=None*, *start_stamp: int = 0*, *end_stamp: Optional[int] = None*, *gcp_service_key: Optional[str] = None*, *interpolate_param: bool = False*, *sort_column=None*, *scaled_cols=None*, *feature_params=None*, *id_series_col=None*, *no_scale=False*)

Bases: Generic[torch.utils.data.dataset.T_co]

__init__(*file_path: str*, *forecast_history: int*, *forecast_length: int*, *target_col: List*, *relevant_cols: List*, *scaling=None*, *start_stamp: int = 0*, *end_stamp: Optional[int] = None*, *gcp_service_key: Optional[str] = None*, *interpolate_param: bool = False*, *sort_column=None*, *scaled_cols=None*, *feature_params=None*, *id_series_col=None*, *no_scale=False*)

A data loader that takes a CSV file and properly batches for use in training/eval a PyTorch model :param file_path: The path to the CSV file you wish to use. :param forecast_history: This is the length of the historical time series data you wish to

utilize for forecasting

Parameters

- **forecast_length** – The number of time steps to forecast ahead (for transformer this must equal history_length)
- **relevant_cols** – Supply column names you wish to predict in the forecast (others will not be used)
- **target_col** – The target column or columns you to predict. If you only have one still use a list ['cfs']
- **scaling** – (highly reccomended) If provided should be a subclass of sklearn.base.BaseEstimator

and sklearn.base.TransformerMixin) i.e StandardScaler, MaxAbsScaler, MinMaxScaler, etc) Note without a scaler the loss is likely to explode and cause infinite loss which will corrupt weights :param start_stamp int: Optional if you want to only use part of a CSV for training, validation

or testing supply these

Parameters

- **int** (*end_stamp*) – Optional if you want to only use part of a CSV for training, validation, or testing supply these
- **str** (*sort_column*) – The column to sort the time series on prior to forecast.
- **scaled_cols** – The columns you want scaling applied to (if left blank will default to all columns)
- **feature_params** – These are the datetime features you want to create.
- **no_scale** – This means that the end labels will not be scaled when running

inverse_scale(*result_data: Union[torch.Tensor, pandas.core.series.Series, numpy.ndarray]*) → torch.Tensor

Un-does the scaling of the data

Parameters **result_data** (*Union[torch.Tensor, pd.Series, np.ndarray]*) – The data you want to unscale can handle multiple data types.

Returns Returns the unscaled data as PyTorch tensor.

Return type torch.Tensor

class flood_forecast.preprocessing.pytorch_loaders.**CSVTestLoader**(*df_path: str*, *forecast_total: int*, *use_real_precip=True*, *use_real_temp=True*, *target_supplied=True*, *interpolate=False*, *sort_column_clone=None*, ***kwargs*)

Bases: Generic[torch.utils.data.dataset.T_co]

__init__(*df_path: str*, *forecast_total: int*, *use_real_precip=True*, *use_real_temp=True*, *target_supplied=True*, *interpolate=False*, *sort_column_clone=None*, ***kwargs*)

Parameters **df_path** (*str*) –

A data loader for the test data.

get_from_start_date(*forecast_start: datetime.datetime*)

convert_real_batches(*the_col: str*, *rows_to_convert*)
A helper function to return properly divided precip and temp values to be stacked with forecasted cfs.

convert_history_batches(*the_col: Union[str, List[str]]*, *rows_to_convert: pandas.core.frame.DataFrame*)
A helper function to return dataframe in batches of size (history_len, num_features)

Args: the_col (str): column names rows_to_convert (pd.Dataframe): rows in a dataframe to be converted into batches

inverse_scale(*result_data: Union[torch.Tensor, pandas.core.series.Series, numpy.ndarray]*) → torch.Tensor
Un-does the scaling of the data

Parameters **result_data** (*Union[torch.Tensor, pd.Series, np.ndarray]*) – The data you want to unscale can handle multiple data types.

Returns Returns the unscaled data as PyTorch tensor.

Return type torch.Tensor

class flood_forecast.preprocessing.pytorch_loaders.**AEDataloader**(*file_path: str*, *relevant_cols: List*, *scaling=None*, *start_stamp: int = 0*, *target_col: Optional[List] = None*, *end_stamp: Optional[int] = None*, *unsqueeze_dim: int = 1*, *interpolate_param=False*, *forecast_history=1*, *no_scale=True*, *sort_column=None*)
Bases: Generic[torch.utils.data.dataset.T_co]

__init__(*file_path: str*, *relevant_cols: List*, *scaling=None*, *start_stamp: int = 0*, *target_col: Optional[List] = None*, *end_stamp: Optional[int] = None*, *unsqueeze_dim: int = 1*, *interpolate_param=False*, *forecast_history=1*, *no_scale=True*, *sort_column=None*)
A data loader class for autoencoders. Overrides __len__ and __getitem__ from generic dataloader. Also defaults forecast_history and forecast_length to 1. Since AE will likely only use one row. Same parameters as before.

inverse_scale(*result_data: Union[torch.Tensor, pandas.core.series.Series, numpy.ndarray]*) → torch.Tensor
Un-does the scaling of the data

Parameters **result_data** (*Union[torch.Tensor, pd.Series, np.ndarray]*) – The data you want to unscale can handle multiple data types.

Returns Returns the unscaled data as PyTorch tensor.

Return type torch.Tensor

class flood_forecast.preprocessing.pytorch_loaders.**TemporalLoader**(*time_feats: List[str]*, *kwargs*)

Bases: Generic[torch.utils.data.dataset.T_co]

__init__(*time_feats: List[str]*, *kwargs*)
[summary]

Parameters

- **time_feats** (*List[str]*) – [description]
- **kwargs** (*[type]*) – [description]

static df_to_numpy(*pandas_stuff: pandas.core.frame.DataFrame*)

inverse_scale(*result_data: Union[torch.Tensor, pandas.core.series.Series, numpy.ndarray]*) → torch.Tensor
Un-does the scaling of the data

Parameters **result_data** (*Union[torch.Tensor, pd.Series, np.ndarray]*) – The data you want to unscale can handle multiple data types.

Returns Returns the unscaled data as PyTorch tensor.

Return type torch.Tensor

class flood_forecast.preprocessing.pytorch_loaders.**TemporalTestLoader**(*time_feats*, *kwargs={}*, *decoder_step_len=None*)

Bases: Generic[torch.utils.data.dataset.T_co]

__init__(*time_feats*, *kwargs={}*, *decoder_step_len=None*)
[summary]

Parameters

- **time_feats** (*[type]*) – [description]
- **kwargs** (*dict, optional*) – [description], defaults to {}
- **decoder_step_len** (*[type], optional*) – [description], defaults to None

convert_history_batches(*the_col: Union[str, List[str]]*, *rows_to_convert: pandas.core.frame.DataFrame*)
A helper function to return dataframe in batches of size (history_len, num_features)

Args: the_col (str): column names rows_to_convert (pd.Dataframe): rows in a dataframe to be converted into batches

convert_real_batches(*the_col: str*, *rows_to_convert*)
A helper function to return properly divided precip and temp values to be stacked with forecasted cfs.

get_from_start_date(*forecast_start: datetime.datetime*)

inverse_scale(*result_data: Union[torch.Tensor, pandas.core.series.Series, numpy.ndarray]*) → torch.Tensor
Un-does the scaling of the data

Parameters **result_data** (*Union[torch.Tensor, pd.Series, np.ndarray]*) – The data you want to unscale can handle multiple data types.

Returns Returns the unscaled data as PyTorch tensor.

Return type torch.Tensor

static df_to_numpy(*pandas_stuff: pandas.core.frame.DataFrame*)

CHAPTER

EIGHTEEN

TEMPORAL FEATURES

flood_forecast.preprocessing.temporal_feats.**make_temporal_features**(*features_list: Dict*, *dt_column: str*, *df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

A function that creates temporal features

Parameters

- **features_list** (*Dict*) – A list of features in the form of a dictionary
- **dt_column** (*str*) – [description]
- **df** (*pd.DataFrame*) – [description]

Returns The DF with several new columns added

Return type pd.DataFrame

flood_forecast.preprocessing.temporal_feats.**create_feature**(*key: str*, *value: str*, *df: pandas.core.frame.DataFrame*, *dt_column: str*)

Function to create temporal features Uses dict to make val.

Parameters

- **key** (*str s*) – The datetime feature you would like to create
- **value** (*str*) – The type of feature you would like to create (cyclical or numerical)
- **df** (*pd.DataFrame*) – The Pandas dataframe with the datetime
- **dt_column** (*str*) – The name of the datetime column

Returns The dataframe with the newly added column

Return type pd.DataFrame

flood_forecast.preprocessing.temporal_feats.**feature_fix**(*preprocess_params: Dict*, *dt_column: str*, *df: pandas.core.frame.DataFrame*)

Adds temporal features

Parameters

- **preprocess_params** (*Dict*) – Dictionary of temporal parameters e.g. {"day":"numerical"}
- **dt_column** – The column name of the data
- **df** (*pd.DataFrame*) – The dataframe to add the temporal features to

Returns Returns the new data-frame and a list of the new column names

Return type Tuple(pd.Dataframe, List[str])

flood_forecast.preprocessing.temporal_feats.**cyclical**(*df: pandas.core.frame.DataFrame*, *feature_column: str*) → pandas.core.frame.DataFrame

A function to create cyclical encodings for Pandas data-frames.

Parameters

- **df** (*pd.DataFrame*) – A Pandas Dataframe where you want the dt encoded
- **feature_column** (*str*) – The name of the feature column. Should be either (day_of_week, hour, month, year)

Returns The dataframe with three new columns: norm_feature, cos_feature, sin_feature

Return type pd.DataFrame

CHAPTER

NINETEEN

CUSTOM OPTIMIZATIONS

flood_forecast.custom.custom_opt.**warmup_cosine**(*x*, *warmup=0.002*)

flood_forecast.custom.custom_opt.**warmup_constant**(*x*, *warmup=0.002*)
Linearly increases learning rate over *warmup`*`t_total* (as provided to BertAdam) training steps. Learning rate is 1. afterwards.

flood_forecast.custom.custom_opt.**warmup_linear**(*x*, *warmup=0.002*)
Specifies a triangular learning rate schedule where peak is reached at *warmup`*`t_total*-th (as provided to BertAdam) training step. After *t_total*-th training step, learning rate is zero.

class flood_forecast.custom.custom_opt.**MASELoss**(*baseline_method*)

__init__(*baseline_method*)
This implements the MASE loss function (e.g. MAE_MODEL/MAE_NAIEVE)

forward(*target: torch.Tensor*, *output: torch.Tensor*, *train_data: torch.Tensor*, *m=1*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**RMSELoss**(*variance_penalty=0.0*)
Returns RMSE using: target -> True y output -> Prediction by model source: https://discuss.pytorch.org/t/rmse-loss-function/16540/3

__init__(*variance_penalty=0.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*output: torch.Tensor*, *target: torch.Tensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**MAPELoss**(*variance_penalty=0.0*)
Returns MAPE using: target -> True y output -> Predtion by model

__init__(*variance_penalty=0.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*output: torch.Tensor*, *target: torch.Tensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**PenalizedMSELoss**(*variance_penalty=0.0*)
Returns MSE using: target -> True y output -> Predtion by model source: https://discuss.pytorch.org/t/rmse-loss-function/16540/3

__init__(*variance_penalty=0.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*output: torch.Tensor*, *target: torch.Tensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**GaussianLoss**(*mu=0*, *sigma=0*)

__init__(*mu=0*, *sigma=0*)
Compute the negative log likelihood of Gaussian Distribution From https://arxiv.org/abs/1907.00235

forward(*x: torch.Tensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**QuantileLoss**(*quantiles*)
From https://medium.com/the-artificial-impostor/quantile-regression-part-2-6fdbc26b2629

__init__(*quantiles*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*preds*, *target*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.custom.custom_opt.**BertAdam**(*params*, *lr=<required parameter>*, *warmup=-1*, *t_total=-1*, *schedule='warmup_linear'*, *b1=0.9*, *b2=0.999*, *e=1e-06*, *weight_decay=0.01*, *max_grad_norm=1.0*)
Implements BERT version of Adam algorithm with weight decay fix. Params:

lr: learning rate warmup: portion of t_total for the warmup, -1 means no warmup. Default: -1 t_total: total number of training steps for the learning

rate schedule, -1 means constant learning rate. Default: -1

schedule: schedule to use for the warmup (see above). Default: 'warmup_linear' b1: Adams b1. Default: 0.9 b2: Adams b2. Default: 0.999 e: Adams epsilon. Default: 1e-6 weight_decay: Weight decay. Default: 0.01 max_grad_norm: Maximum norm for the gradients (-1 means no clipping). Default: 1.0

__init__(*params*, *lr=<required parameter>*, *warmup=-1*, *t_total=-1*, *schedule='warmup_linear'*, *b1=0.9*, *b2=0.999*, *e=1e-06*, *weight_decay=0.01*, *max_grad_norm=1.0*)
Initialize self. See help(type(self)) for accurate signature.

get_lr() → List

step(*closure=None*)
Performs a single optimization step. Arguments:

closure (callable, optional): A closure that reevaluates the model and returns the loss.

class flood_forecast.custom.custom_opt.**NegativeLogLikelihood**
target -> True y output -> predicted distribution

__init__()
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*output: <module 'torch.distributions' from '/home/docs/checkouts/readthedocs.org/user_builds/flow-forecast/envs/stable/lib/python3.7/site-packages/torch/distributions/__init__.py'>*, *target: torch.Tensor*)
calculates NegativeLogLikelihood

training: bool

flood_forecast.custom.custom_opt.**l1_regularizer**(*model*, *lambda_l1=0.01*)
source: https://stackoverflow.com/questions/58172188/how-to-add-l1-regularization-to-pytorch-nn-model

flood_forecast.custom.custom_opt.**orth_regularizer**(*model*, *lambda_orth=0.01*)
source: https://stackoverflow.com/questions/58172188/how-to-add-l1-regularization-to-pytorch-nn-model

sss

CHAPTER

TWENTY

DUMMY TORCH MODEL

A dummy model specifically for unit and integration testing purposes

class flood_forecast.transformer_xl.dummy_torch.**DummyTorchModel**(*forecast_length: int*)

__init__(*forecast_length: int*)

A dummy model that will return a tensor of ones (batch_size, forecast_len)

Parameters **forecast_length** (*int*) – The length to forecast

forward(*x: torch.Tensor*) → torch.Tensor

The forward pass for the dummy model :param x: Here the data is irrelvant. Only batch_size is grabbed :type x: torch.Tensor :param mask: [description], defaults to None :type mask: [type], optional :return: [description] :rtype: torch.Tensor

training: bool

CHAPTER

TWENTYONE

LOWER UPPER CONFIGURATION

flood_forecast.transformer_xl.lower_upper_config.**initial_layer**(*layer_type: str*, *layer_params: Dict*, *layer_number: int = 1*)

flood_forecast.transformer_xl.lower_upper_config.**swish**(*x*)

flood_forecast.transformer_xl.lower_upper_config.**variable_forecast_layer**(*layer_type*, *layer_params*)

class flood_forecast.transformer_xl.lower_upper_config.**PositionwiseFeedForward**(*d_in*, *d_hid*, *dropout=0.1*)

A two-feed-forward-layer module Taken from DSANET repos

__init__(*d_in*, *d_hid*, *dropout=0.1*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.lower_upper_config.**AR**(*window*)

__init__(*window*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.lower_upper_config.**MetaEmbedding**(*meta_vector_dim: int*, *output_dim*, *predictor_number: int*, *predictor_order*)

__init__(*meta_vector_dim: int*, *output_dim*, *predictor_number: int*, *predictor_order*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

training: bool

CHAPTER

TWENTYTWO

SIMPLE MULTI ATTENTION HEAD MODEL

class flood_forecast.transformer_xl.multi_head_base.**MultiAttnHeadSimple**(*number_time_series: int*, *seq_len=10*, *output_seq_len=None*, *d_model=128*, *num_heads=8*, *forecast_length=None*, *dropout=0.1*, *final_layer=False*)

A simple multi-head attention model inspired by Vaswani et al.

__init__(*number_time_series: int*, *seq_len=10*, *output_seq_len=None*, *d_model=128*, *num_heads=8*, *forecast_length=None*, *dropout=0.1*, *final_layer=False*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x: torch.Tensor*, *mask=None*) → torch.Tensor

Param x torch.Tensor: of shape (B, L, M)

Where B is the batch size, L is the sequence length and M is the number of time :return: a tensor of dimension (B, forecast_length)

training: bool

CHAPTER

TWENTYTHREE

BASIC TRANSFORMER

class flood_forecast.transformer_xl.transformer_basic.**SimpleTransformer**(*number_time_series: int*, *seq_length: int = 48*, *output_seq_len: Optional[int] = None*, *d_model: int = 128*, *n_heads: int = 8*, *dropout=0.1*, *forward_dim=2048*, *sigmoid=False*)

__init__(*number_time_series: int*, *seq_length: int = 48*, *output_seq_len: Optional[int] = None*, *d_model: int = 128*, *n_heads: int = 8*, *dropout=0.1*, *forward_dim=2048*, *sigmoid=False*)

Full transformer model

forward(*x: torch.Tensor*, *t: torch.Tensor*, *tgt_mask=None*, *src_mask=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

basic_feature(*x: torch.Tensor*)

encode_sequence(*x*, *src_mask=None*)

decode_seq(*mem*, *t*, *tgt_mask=None*, *view_number=None*) → torch.Tensor

training: bool

class flood_forecast.transformer_xl.transformer_basic.**CustomTransformerDecoder**(*seq_length: int*, *output_seq_length: int*, *n_time_series: int*, *d_model=128*, *output_dim=1*, *n_layers_encoder=6*, *forward_dim=2048*, *dropout=0.1*, *use_mask=False*, *meta_data=None*, *final_act=None*, *n_heads=8*)

__init__(*seq_length: int*, *output_seq_length: int*, *n_time_series: int*, *d_model=128*, *output_dim=1*, *n_layers_encoder=6*, *forward_dim=2048*, *dropout=0.1*, *use_mask=False*, *meta_data=None*, *final_act=None*, *n_heads=8*)

Uses a number of encoder layers with simple linear decoder layer.

make_embedding(*x: torch.Tensor*)

forward(*x: torch.Tensor*, *meta_data=None*) → torch.Tensor

Performs forward pass on tensor of (batch_size, sequence_length, n_time_series) Return tensor of dim (batch_size, output_seq_length)

training: bool

class flood_forecast.transformer_xl.transformer_basic.**SimplePositionalEncoding**(*d_model*, *dropout=0.1*, *max_len=5000*)

__init__(*d_model*, *dropout=0.1*, *max_len=5000*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x: torch.Tensor*) → torch.Tensor

Creates a basic positional encoding

training: bool

flood_forecast.transformer_xl.transformer_basic.**greedy_decode**(*model*, *src: torch.Tensor*, *max_len: int*, *real_target: torch.Tensor*, *unsqueeze_dim=1*, *output_len=1*, *device='cpu'*, *multi_targets=1*, *probabilistic=False*, *scaler=None*)

Mechanism to sequentially decode the model :src The Historical time series values :real_target The real values (they should be masked), however if you want can include known real values. :returns torch.Tensor

CHAPTER

TWENTYFOUR

TRANSFORMER XL

Model from Keita Kurita. Not useable https://github.com/keitakurita/Practical_NLP_in_PyTorch/blob/master/deep_dives/transformer_xl_from_scratch.ipynb

class flood_forecast.transformer_xl.transformer_xl.**MultiHeadAttention**(*d_input: int*, *d_inner: int*, *n_heads: int = 4*, *dropout: float = 0.1*, *dropouta: float = 0.0*)

__init__(*d_input: int*, *d_inner: int*, *n_heads: int = 4*, *dropout: float = 0.1*, *dropouta: float = 0.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_: torch.FloatTensor*, *pos_embs: torch.FloatTensor*, *memory: torch.FloatTensor*, *u: torch.FloatTensor*, *v: torch.FloatTensor*, *mask: Optional[torch.FloatTensor] = None*)

pos_embs: we pass the positional embeddings in separately because we need to handle relative positions

input shape: (seq, bs, self.d_input) pos_embs shape: (seq + prev_seq, bs, self.d_input) output shape: (seq, bs, self.d_input)

training: bool

class flood_forecast.transformer_xl.transformer_xl.**PositionwiseFF**(*d_input*, *d_inner*, *dropout*)

__init__(*d_input*, *d_inner*, *dropout*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_: torch.FloatTensor*) → torch.FloatTensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the

Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class flood_forecast.transformer_xl.transformer_xl.**DecoderBlock**(*n_heads*, *d_input*, *d_head_inner*, *d_ff_inner*, *dropout*, *dropouta=0.0*)

__init__(*n_heads*, *d_input*, *d_head_inner*, *d_ff_inner*, *dropout*, *dropouta=0.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_: torch.FloatTensor*, *pos_embs: torch.FloatTensor*, *u: torch.FloatTensor*, *v: torch.FloatTensor*, *mask=None*, *mems=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class flood_forecast.transformer_xl.transformer_xl.**PositionalEmbedding**(*d*)

__init__(*d*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*positions: torch.LongTensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class flood_forecast.transformer_xl.transformer_xl.**StandardWordEmbedding**(*num_embeddings*, *embedding_dim*, *div_val=1*, *sample_softmax=False*)

__init__(*num_embeddings*, *embedding_dim*, *div_val=1*, *sample_softmax=False*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_: torch.LongTensor*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_xl.**TransformerXL**(*num_embeddings*, *n_layers*, *n_heads*, *d_model*, *d_head_inner*, *d_ff_inner*, *dropout=0.1*, *dropouta=0.0*, *seq_len: int = 0*, *mem_len: int = 0*)

__init__(*num_embeddings*, *n_layers*, *n_heads*, *d_model*, *d_head_inner*, *d_ff_inner*, *dropout=0.1*, *dropouta=0.0*, *seq_len: int = 0*, *mem_len: int = 0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

init_memory(*device=device(type='cpu')*) → torch.FloatTensor

update_memory(*previous_memory: List[torch.FloatTensor]*, *hidden_states: List[torch.FloatTensor]*)

reset_length(*seq_len*, *ext_len*, *mem_len*)

forward(*idxs: torch.LongTensor*, *target: torch.LongTensor*, *memory: Optional[List[torch.FloatTensor]] = None*) → Dict[str, torch.Tensor]
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

CHAPTER

TWENTYFIVE

CONVOLUTIONAL TRANSFORMER

This is an implementation of the code from this paper

This code is based on huggingface, https://github.com/huggingface/pytorch-openai-transformer-lm/blob/master/model_pytorch.py

flood_forecast.transformer_xl.transformer_bottleneck.**gelu**(*x*)

flood_forecast.transformer_xl.transformer_bottleneck.**swish**(*x*)

class flood_forecast.transformer_xl.transformer_bottleneck.**Attention**(*n_head*, *n_embd*, *win_len*, *scale*, *q_len*, *sub_len*, *sparse=None*, *attn_pdrop=0.1*, *resid_pdrop=0.1*)

__init__(*n_head*, *n_embd*, *win_len*, *scale*, *q_len*, *sub_len*, *sparse=None*, *attn_pdrop=0.1*, *resid_pdrop=0.1*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

log_mask(*win_len*, *sub_len*)

row_mask(*index*, *sub_len*, *win_len*)
Remark: 1 . Currently, dense matrices with sparse multiplication are not supported by Pytorch. Efficient implementation

should deal with CUDA kernel, which we haven't implemented yet.

2 . Our default setting here use Local attention and Restart attention.

3 . For index-th row, if its past is smaller than the number of cells the last cell can attend, we can allow current cell to attend all past cells to fully utilize parallel computing in dense matrices with sparse multiplication.

attn(*query: torch.Tensor*, *key*, *value: torch.Tensor*, *activation='Softmax'*)

merge_heads(*x*)

split_heads(*x*, *k=False*)

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**Conv1D**(*out_dim*, *rf*, *in_dim*)

__init__(*out_dim*, *rf*, *in_dim*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**LayerNorm**(*n_embd*, *e=1e-05*)

Construct a layernorm module in the OpenAI style (epsilon inside the square root).

__init__(*n_embd*, *e=1e-05*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**MLP**(*n_state*, *n_embd*, *acf='relu'*)

__init__(*n_state*, *n_embd*, *acf='relu'*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**Block**(*n_head*, *win_len*, *n_embd*, *scale*, *q_len*, *sub_len*, *additional_params: Dict*)

__init__(*n_head*, *win_len*, *n_embd*, *scale*, *q_len*, *sub_len*, *additional_params: Dict*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**TransformerModel**(*n_time_series*, *n_head*, *sub_len*, *num_layer*, *n_embd*, *forecast_history: int*, *dropout: float*, *scale_att*, *q_len*, *additional_params: Dict*, *seq_num=None*)

Transformer model

__init__(*n_time_series*, *n_head*, *sub_len*, *num_layer*, *n_embd*, *forecast_history: int*, *dropout: float*, *scale_att*, *q_len*, *additional_params: Dict*, *seq_num=None*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

win_len

For positional encoding in Transformer, we use learnable position embedding. For covariates, following [3], we use all or part of year, month, day-of-the-week, hour-of-the-day, minute-of-the-hour, age and time-series-ID according to the granularities of datasets. age is the distance to the first observation in that time series [3]. Each of them except time series ID has only one dimension and is normalized to have zero mean and unit variance (if applicable).

forward(*series_id: int*, *x: torch.Tensor*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.transformer_bottleneck.**DecoderTransformer**(*n_time_series: int*, *n_head: int*, *num_layer: int*, *n_embd: int*, *forecast_history: int*, *dropout: float*, *q_len: int*, *additional_params: Dict*, *activation='Softmax'*, *forecast_length: Optional[int] = None*, *scale_att: bool = False*, *seq_num=None*, *sub_len=1*, *mu=None*)

__init__(*n_time_series: int*, *n_head: int*, *num_layer: int*, *n_embd: int*, *forecast_history: int*, *dropout: float*, *q_len: int*, *additional_params: Dict*, *activation='Softmax'*, *forecast_length: Optional[int] = None*, *scale_att: bool = False*, *seq_num=None*, *sub_len=1*, *mu=None*)

Args: n_time_series: Number of time series present in input n_head: Number of heads in the MultiHeadAttention mechanism seq_num: The number of targets to forecast sub_len: sub_len of the sparse attention num_layer: The number of transformer blocks in the model. n_embd: The dimention of Position embedding and time series ID embedding forecast_history: The number of historical steps fed into the time series model dropout: The dropout for the embedding of the model. additional_params: Additional parameters used to initalize the attention model. Can inc

training: bool

forward(*x: torch.Tensor*, *series_id: Optional[int] = None*)

Args: x: Tensor of dimension (batch_size, seq_len, number_of_time_series) series_id: Optional id of the series in the dataframe. Currently not supported

Returns: Case 1: tensor of dimension (batch_size, forecast_length) Case 2: GLoss sigma and mu: tuple of ((batch_size, forecast_history, 1), (batch_size, forecast_history, 1))

CHAPTER

TWENTYSIX

INFORMER

class flood_forecast.transformer_xl.informer.**Informer**(*n_time_series: int*, *dec_in: int*, *c_out: int*, *seq_len*, *label_len*, *out_len*, *factor=5*, *d_model=512*, *n_heads=8*, *e_layers=3*, *d_layers=2*, *d_ff=512*, *dropout=0.0*, *attn='prob'*, *embed='fixed'*, *temp_depth=4*, *activation='gelu'*, *device=device(type='cuda', index=0)*)

__init__(*n_time_series: int*, *dec_in: int*, *c_out: int*, *seq_len*, *label_len*, *out_len*, *factor=5*, *d_model=512*, *n_heads=8*, *e_layers=3*, *d_layers=2*, *d_ff=512*, *dropout=0.0*, *attn='prob'*, *embed='fixed'*, *temp_depth=4*, *activation='gelu'*, *device=device(type='cuda', index=0)*)

This is based on the implementation of the Informer available from the original authors https://github.com/zhouhaoyi/Informer2020. We have done some minimal refactoring, but the core code remains the same.

Parameters

- **n_time_series** (*int*) – The number of time series present in the multivariate forecasting problem.
- **dec_in** (*int*) – The input size to the decoder (e.g. the number of time series passed to the decoder)
- **c_out** (*int*) – The output dimension of the model (usually will be the number of variables you are forecasting).
- **seq_len** (*int*) – The number of historical time steps to pass into the model.
- **label_len** (*int*) – The length of the label sequence passed into the decoder.
- **out_len** (*int*) – The overall output length from the decoder .
- **factor** (*int, optional*) – The multiplicative factor in the probablistic attention mechanism, defaults to 5
- **d_model** (*int, optional*) – The embedding dimension of the model, defaults to 512
- **n_heads** (*int, optional*) – The number of heads in the multi-head attention mechanism , defaults to 8
- **e_layers** (*int, optional*) – The number of layers in the encoder, defaults to 3

- **d_layers** (*int, optional*) – The number of layers in the decoder, defaults to 2
- **d_ff** (*int, optional*) – The dimension of the forward pass, defaults to 512
- **dropout** (*float, optional*) – [description], defaults to 0.0
- **attn** (*str, optional*) – The type of the attention mechanism either 'prob' or 'full', defaults to 'prob'
- **embed** (*str, optional*) – Whether to use class: *FixedEmbedding* or *torch.nn.Embbeding* , defaults to 'fixed'
- **temp_depth** – The temporald depth (e.g), defaults to 4
- **activation** (*str, optional*) – The activation func, defaults to 'gelu'
- **device** (*str, optional*) – The device the model uses, defaults to torch.device('cuda:0')

forward(*x_enc*, *x_mark_enc*, *x_dec*, *x_mark_dec*, *enc_self_mask=None*, *dec_self_mask=None*, *dec_enc_mask=None*)

Parameters

- **x_enc** (*torch.Tensor*) – The core tensor going into the model. Of dimension (batch_size, seq_len, n_time_series)
- **x_mark_enc** (*torch.Tensor*) – A tensor with the relevant datetime information. (batch_size, seq_len, n_datetime_feats)
- **x_dec** (*torch.Tensor*) – The datetime tensor information. Has dimension batch_size, seq_len, n_time_series
- **x_mark_dec** (*torch.Tensor*) – A tensor with the relevant datetime information. (batch_size, seq_len, n_datetime_feats)
- **enc_self_mask** (*[type], optional*) – The mask of the encoder model has size (), defaults to None
- **dec_self_mask** (*[type], optional*) – [description], defaults to None
- **dec_enc_mask** (*[type], optional*) – [description], defaults to None

Returns Returns a PyTorch tensor of shape (batch_size, ?, ?)

Return type torch.Tensor

training: bool

class flood_forecast.transformer_xl.informer.**ConvLayer**(*c_in*)

__init__(*c_in*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.informer.**EncoderLayer**(*attention*, *d_model*, *d_ff=None*, *dropout=0.1*, *activation='relu'*)

__init__(*attention*, *d_model*, *d_ff=None*, *dropout=0.1*, *activation='relu'*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *attn_mask=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.informer.**Encoder**(*attn_layers*, *conv_layers=None*, *norm_layer=None*)

__init__(*attn_layers*, *conv_layers=None*, *norm_layer=None*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *attn_mask=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.informer.**DecoderLayer**(*self_attention*, *cross_attention*, *d_model*, *d_ff=None*, *dropout=0.1*, *activation='relu'*)

__init__(*self_attention*, *cross_attention*, *d_model*, *d_ff=None*, *dropout=0.1*, *activation='relu'*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *cross*, *x_mask=None*, *cross_mask=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class flood_forecast.transformer_xl.informer.**Decoder**(*layers*, *norm_layer=None*)

__init__(*layers*, *norm_layer=None*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *cross*, *x_mask=None*, *cross_mask=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

CHAPTER

TWENTYSEVEN

BASIC GCP UTILS

flood_forecast.gcp_integration.basic_utils.**get_storage_client**(*service_key_path: Optional[str] = None*) → google.cloud.storage.client.Client

Utility function to return a properly authenticated GCS storage client whether working in Colab, CircleCI, Dataverse, or other environments.

flood_forecast.gcp_integration.basic_utils.**upload_file**(*bucket_name: str*, *file_name: str*, *upload_name: str*, *client: google.cloud.storage.client.Client*)

A function to upload a file to a GCP bucket.

Parameters

- **bucket_name** (*str*) – The name of the bucket
- **file_name** (*str*) – [description]
- **upload_name** (*str*) – [description]
- **client** (*storage.Client*) – [description]

flood_forecast.gcp_integration.basic_utils.**download_file**(*bucket_name: str*, *source_blob_name: str*, *destination_file_name: str*, *service_key_path: Optional[str] = None*) → None

Download data file from GCS.

Args: bucket_name ([str]): bucket name on GCS, eg. task_ts_data source_blob_name ([str]): storage object name destination_file_name ([str]): filepath to save to local

flood_forecast.explain_model_output.**handle_dl_output**(*dl*, *dl_class: str*, *datetime_start: datetime.datetime*, *device: str*) → Tuple[torch.Tensor, int]

Parameters

- **dl** (*Union[CSVTestLoader, TemporalTestLoader]*) – The test data-loader. Should be passed directly
- **dl_class** (*str*) – A string that is the name of DL passef from the params file.
- **datetime_start** (*datetime*) – The start datetime for the forecast
- **device** (*str*) – Typical device should be either cpu or cuda

Returns Returns a tuple containing either a..

Return type Tuple[torch.Tensor, int]

flood_forecast.explain_model_output.**deep_explain_model_summary_plot**(*model*, *csv_test_loader: flood_forecast.preprocessing.pytc*, *date-time_start: Op-tional[datetime.datetime] = None*) → None

Generate feature summary plot for trained deep learning models Args:

model (object): trained model csv_test_loader (CSVTestLoader): test data loader datetime_start (datetime, optional): start date of the test prediction,

Defaults to None, i.e. using model inference parameters.

flood_forecast.explain_model_output.**fix_shap_values**(*shap_values*, *history*)

flood_forecast.explain_model_output.**deep_explain_model_heatmap**(*model*, *csv_test_loader: flood_forecast.preprocessing.pytorch_loa*, *date-time_start: Op-tional[datetime.datetime] = None*) → None

Generate feature heatmap for prediction at a start time Args:

model ([type]): trained model csv_test_loader ([CSVTestLoader]): test data loader datetime_start (Optional[datetime], optional): start date of the test prediction,

Defaults to None, i.e. using model inference parameters.

Returns: None

CHAPTER

TWENTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

E

F

G

H

I

J

L

M

N

O

P

U

V

W