
Flow Forecast

Release 0.0.1

Aug 11, 2020

1	Utilities	1
2	Model Evaluator	3
3	Long Train	7
4	Model Dictionaries	9
5	Pre Dictionaries	11
6	PyTorch Training	13
7	Time Model	15
8	Trainer	17
9	Interpolate Preprocessing	19
10	Build Dataset	21
11	Closest Station	23
12	Data Converter	25
13	Preprocess DA RNN	27
14	Preprocess Metadata	29
15	Process USGS	31
16	PyTorch Loaders	33
17	Temporal Features	35
18	Custom Optimizations	37
19	Dummy Torch Model	69
20	Lower Upper Configuration	77

21 Simple Multi Attention Head Model	101
22 Basic Transformer	111
23 Transformer XL	137
24 Basic GCP Utils	185
25 Train da	187
26 Utils	189
27 Custom Types	191
28 Model	193
29 Modules	203
30 Indices and tables	219
Python Module Index	221
Index	223

`flood_forecast.utils.flatten_list_function(input_list)`

class `flood_forecast.utils.EarlyStopper` (*patience: int, min_delta: float = 0.0, cumulative_delta: bool = False*)

Bases: `object`

EarlyStopping handler can be used to stop the training if no improvement after a given number of events. Args:

patience (int): Number of events to wait if no improvement and then stop the training.

score_function (callable): It should be a function taking a single argument, an `Engine` object, and return a score *float*. An improvement is considered if the score is higher.

trainer (Engine): trainer engine to stop the run if no improvement.

min_delta (float, optional): A minimum increase in the score to qualify as an improvement, i.e. an increase of less than or equal to *min_delta*, will count as no improvement.

cumulative_delta (bool, optional): If True, *min_delta* defines an increase since the last *patience* reset, otherwise, it defines an increase after the last event. Default value is False.

Examples: .. code-block:: python

```
from ignite.engine import Engine, Events
from ignite.handlers import EarlyStopping
def score_function(engine):
```

```
    val_loss = engine.state.metrics['nll']
    return -val_loss
```

```
handler = EarlyStopping(patience=10, score_function=score_function, trainer=trainer) # Note:
the handler is attached to an Evaluator (runs one epoch on validation dataset). evaluator.add_event_handler(Events.COMPLETED, handler)
```

check_loss (*model, validation_loss*) → bool

save_model_checkpoint (*model*)

CHAPTER 2

Model Evaluator

```
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.metric_dict(metric: str) → Callable
```

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

A function to evaluate a model. Requires a model of type TimeSeriesModel

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

Function to handle both test evaluation and inference on a test dataframe. :returns

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

```
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[KT, VT]) ->
                                              torch.Tensor
```

```
flood_forecast.evaluator.generate_predictions_non_decoded(model:
                                                          Type[flood_forecast.time_model.TimeSeriesModel],
                                                          df: pandas
                                                          .core.frame.DataFrame,
                                                          test_data:
                                                          flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader,
                                                          history_dim:
                                                          torch.Tensor, forecast_length:
                                                          int,
                                                          hours_to_forecast: int)
                                                          -> torch.Tensor
```

```
flood_forecast.evaluator.generate_decoded_predictions(model:
                                                         Type[flood_forecast.time_model.TimeSeriesModel],
                                                         test_data:
                                                         flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader,
                                                         forecast_start_idx: int,
                                                         device: torch.device, history_dim:
                                                         torch.Tensor,
                                                         hours_to_forecast: int, decoder_params:
                                                         Dict[KT, VT]) -> 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, de-
                                                    coder_params: Dict[KT, VT],
                                                    num_prediction_samples: int)
                                                    → numpy.ndarray
```


Long Train

```
flood_forecast.long_train.split_on_letter(s: str) → List[T]
```

[illegible]

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

```
flood_forecast.long_train.make_config_file(flow_file_path: str, gage_id: str, station_id: str, weight_path=None)
```

```
flood_forecast.long_train.main()
```


CHAPTER 4

Model Dictionaries

`flood_forecast.model_dict_function.generate_square_subsequent_mask` (*sz: int*) → `torch.Tensor`
Generate a square mask for the sequence. The masked positions are filled with `float('-inf')`. Unmasked positions are filled with `float(0.0)`.

CHAPTER 5

Pre Dictionaries

PyTorch Training

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

Function to train any PyTorchForecast model :model The initialized PyTorchForecastModel :training_params_dict A dictionary of the parameters needed to train model :takes_target boolean: Determines whether to pass target during training :forward_params: A dictionary for additional forward parameters (for instance target)

```
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, for-
    ward_params: Dict[KT, VT]
    = {}) → float
```

```
flood_forecast.pytorch_training.compute_validation(validation_loader:  
                                                    torch.utils.data.dataloader.DataLoader;  
                                                    model, epoch: int, se-  
                                                    quence_size: int, criterion:  
                                                    Type[torch.nn.modules.loss._Loss],  
                                                    device: torch.device, de-  
                                                    coder_structure=False,  
                                                    use_wandb: bool = False,  
                                                    val_or_test='validation_loss') →  
                                                    float
```

Function to compute the validation or test loss

CHAPTER 7

Time Model

```
class flood_forecast.time_model.TimeSeriesModel(model_base: str, training_data: str,  
                                                validation_data: str, test_data: str,  
                                                params: Dict[KT, VT])
```

Bases: `abc.ABC`

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.

load_model (*model_base: str, model_params: Dict[KT, VT], weight_path=None*) → object

This function should load and return the model this will vary based on the underlying framework used

make_data_load (*data_path, params: Dict[KT, VT], loader_type: str*) → object

Initializes 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.

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, val-  
                                                idation_data, test_data, params_dict:  
                                                Dict[KT, VT])
```

Bases: `flood_forecast.time_model.TimeSeriesModel`

load_model (*model_base: str, model_params: Dict[KT, VT], weight_path: 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

upload_gcs (*save_path: str, name: str, file_type: str, epoch=0, bucket_name=None*)

Function to upload model checkpoints to GCS

wandb_init ()

make_data_loader (*data_path*: str, *dataset_params*: Dict[KT, VT], *loader_type*: str,
 the_class='default')

Initializes 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.trainer.train_function(model_type: str, params: Dict[KT, VT])`

Function to train a Model(TimeSeriesModel) or da_rnn. Will return the trained model
model_type str: Type of the model (for now) must be da_rnn or :params dict: Dictionary containing all the parameters needed to run the model

`flood_forecast.trainer.main()`

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

Interpolate Preprocessing

`flood_forecast.preprocessing.interpolate_preprocess.fix_timezones` (*csv_path*:
str) → `pan-`
`das.core.frame.DataFrame`
 Basic function to fix initial data bug related to NaN values in non-eastern-time zones due to UTC conversion.

`flood_forecast.preprocessing.interpolate_preprocess.split_on_na_chunks` (*df*:
`pan-`
`das.core.frame.DataFrame`)
 →
 None

`flood_forecast.preprocessing.interpolate_preprocess.interpolate_missing_values` (*df*:
`pan-`
`das.core.frame.D`
 →
`pan-`
`das.core.frame.D`
 Function to fill missing values with nearest value. Should be run only after splitting on the NaN chunks.

CHAPTER 10

Build Dataset

```
flood_forecast.preprocessing.buil_dataset.build_weather_csv(json_full_path,  
                                                             asos_base_url,  
                                                             base_url_2,  
                                                             econet_data,    vis-  
                                                             ited_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:          pan-  
                                                         das.core.frame.DataFrame,  
                                                         precip_df:          pan-  
                                                         das.core.frame.DataFrame)
```

```
flood_forecast.preprocessing.buil_dataset.create_usgs(meta_data_dir:  str,  pre-  
                                                         cip_path: str, start: int, end:  
                                                         int)
```


CHAPTER 11

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[T],
                                                              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) → date-time.datetime
```

```
flood_forecast.preprocessing.closest_station.convert_temp(temperature: 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[KT, VT]
```

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 12

Data Converter

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 13

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 14

Preprocess Metadata

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


CHAPTER 15

Process USGS

```
flood_forecast.preprocessing.process_usgs.make_usgs_data(start_date:      date-  
                                                         time.datetime, end_date:  
                                                         datetime.datetime,  
                                                         site_number:  
                                                         str)      →      pan-  
                                                         das.core.frame.DataFrame  
flood_forecast.preprocessing.process_usgs.process_response_text(file_name:  
                                                         str)      →  
                                                         Tuple[str,  
                                                         Dict[KT, VT]]  
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 file  
flood_forecast.preprocessing.process_usgs.get_timezone_map()  
flood_forecast.preprocessing.process_usgs.process_intermediate_csv(df:  pan-  
                                                         das.core.frame.DataFrame)  
                                                         -> (<class  
                                                         'pan-  
                                                         das.core.frame.DataFrame'>,  
                                                         <class  
                                                         'int'>,  
                                                         <class  
                                                         'int'>,  
                                                         <class  
                                                         'int'>)
```



```
class flood_forecast.preprocessing.pytorch_loaders.CSVDataLoader (file_path:
                                                                    str,      fore-
                                                                    cast_history:
                                                                    int,      fore-
                                                                    cast_length:
                                                                    int,      tar-
                                                                    get_col:
                                                                    List[T], rel-
                                                                    evant_cols:
                                                                    List[T], scal-
                                                                    ing=None,
                                                                    start_stamp:
                                                                    int      = 0,
                                                                    end_stamp:
                                                                    int      = None,
                                                                    interpo-
                                                                    late_param=True)
```

Bases: torch.utils.data.dataset.Dataset

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 recommended) 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

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

```
class flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader (df_path:
                                                                    str,      fore-
                                                                    cast_total:
                                                                    int,
                                                                    use_real_precip=True,
                                                                    use_real_temp=True,
                                                                    tar-
                                                                    get_supplied=True,
                                                                    interpo-
                                                                    late=False,
                                                                    **kwargs)
```

Bases: `flood_forecast.preprocessing.pytorch_loaders.CSVDataLoader`

Parameters `df_path (str)` –

A data loader for the test data.

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

get_from_start_date (`forecast_start: int`)

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

CHAPTER 17

Temporal Features

```
flood_forecast.preprocessing.temporal_feats.make_temporal_features (features_list:  
                                                                    Dict[KT,  
                                                                    VT],  
                                                                    dt_column:  
                                                                    str, df:  
                                                                    pan-  
                                                                    das.core.frame.DataFrame)  
                                                                    → pan-  
                                                                    das.core.frame.DataFrame
```

Function to create features

```
flood_forecast.preprocessing.temporal_feats.get_day (x: datetime.datetime) → int  
flood_forecast.preprocessing.temporal_feats.get_month (x: datetime.datetime)  
flood_forecast.preprocessing.temporal_feats.get_hour (x: datetime.datetime)  
flood_forecast.preprocessing.temporal_feats.get_weekday (x: datetime.datetime)
```


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.RMSELoss`

Bases: `torch.nn.modules.module.Module`

Returns RMSE using: target -> True y output -> Predtion by model source: <https://discuss.pytorch.org/t/rmse-loss-function/16540/3>

forward (*target: torch.Tensor, output: torch.Tensor*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
    print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```

>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())

```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```

>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that

will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
 Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None
 Sets gradients of all model parameters to zero.

class `flood_forecast.custom.custom_opt.MAPELoss`
 Bases: `torch.nn.modules.module.Module`

Returns MAPE using: target -> True y output -> Predtion by model

forward (*target: torch.Tensor, output: torch.Tensor*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
 Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
(1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double()** → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False****eval()** → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's *state_dict()* function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's *state_dict()* function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
```

(continues on next page)

(continued from previous page)

```

        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

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

Bases: torch.nn.modules.module.Module

Compute the negative log likelihood of Gaussian Distribution From <https://arxiv.org/abs/1907.00235>

forward (*x*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as *self*. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: *self*

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: *self*

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu () → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (device: Union[int, torch.device, None] = None) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: True

Returns:**NamedTuple with missing_keys and unexpected_keys fields:**

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)

```

named_buffers (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```

>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())

```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```

>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)

```

named_modules (memo: Optional[Set[Module]] = None, prefix: str = "")

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, grad_input and grad_output will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use torch.Tensor.register_hook() directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

`name (string)`: name of the buffer. The buffer can be accessed from this module using the given name

`tensor (Tensor)`: buffer to be registered. **`persistent (bool)`:** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad() → None
Sets gradients of all model parameters to zero.

class flood_forecast.custom.custom_opt.QuantileLoss (*quantiles*)
Bases: torch.nn.modules.module.Module

From <https://medium.com/the-artificial-impostor/quantile-regression-part-2-6fdb26b2629>

forward (*preds, target*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if **True**, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double()** → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False**

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self**extra_repr()** → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self**half()** → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self**load_state_dict** (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:**state_dict (dict):** a dict containing parameters and persistent buffers.**strict (bool, optional):** whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True**Returns:****NamedTuple with missing_keys and unexpected_keys fields:**

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if **True**, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. persistent (bool): whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (name: str, param: torch.nn.parameter.Parameter) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (requires_grad: bool = True) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

```
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)
```

Bases: torch.optim.optimizer.Optimizer

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

get_lr () → List[T]

step (closure=None)

Performs a single optimization step. Arguments:

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

add_param_group (param_group)

Add a param group to the Optimizer's `param_groups`.

This can be useful when fine tuning a pre-trained network as frozen layers can be made trainable and added to the Optimizer as training progresses.

Arguments: `param_group` (dict): Specifies what Tensors should be optimized along with group specific optimization options.

load_state_dict (state_dict)

Loads the optimizer state.

Arguments:

state_dict (dict): optimizer state. Should be an object returned from a call to `state_dict()`.

state_dict ()

Returns the state of the optimizer as a dict.

It contains two entries:

- **state - a dict holding current optimization state. Its content** differs between optimizer classes.
- `param_groups` - a dict containing all parameter groups

zero_grad ()

Clears the gradients of all optimized `torch.Tensors`.

CHAPTER 19

Dummy Torch Model

A dummy model specifically for unit and integration testing purposes

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

Bases: torch.nn.modules.module.Module

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

T_destination = ~T_destination

add_module (*name*: str, *module*: torch.nn.modules.module.Module) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn*: Callable[[Module], None]) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()  
>>> def init_weights(m):  
>>>     print(m)  
>>>     if type(m) == nn.Linear:  
>>>         m.weight.fill_(1.0)  
>>>         print(m.weight)
```

(continues on next page)

(continued from previous page)

```

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu**() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self

double() → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
    print(idx, '->', m)
```

(continues on next page)

(continued from previous page)

```

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)

```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```

>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())

```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```

>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)

```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix* (str): prefix to prepend to all parameter names. *recurse* (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse*: bool = True) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook*: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name*: str, *tensor*: torch.Tensor, *persistent*: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → `None`

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: `True`.

Returns: Module: `self`

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (dst_type: Union[torch.dtype, str]) → T

Casts all parameters and buffers to dst_type.

Arguments: dst_type (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

Lower Upper Configuration

```
flood_forecast.transformer_xl.lower_upper_config.initial_layer(layer_type: str,
                                                                layer_params:
                                                                Dict[KT, VT],
                                                                layer_number:
                                                                int = 1)

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)
```

Bases: torch.nn.modules.module.Module

A two-feed-forward-layer module Take from DSANET

forward (*x*)

T_destination = ~**T_destination**

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
 Adds a child module to the current module.
 The module can be accessed as an attribute using the given name.

Args:

- name (string): name of the child module. The child module can be** accessed from this module using the given name
- module (Module):** child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
 Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
    print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```

>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())

```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```

>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that

will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None
Sets gradients of all model parameters to zero.

class `flood_forecast.transformer_xl.lower_upper_config.AR` (*window*)
Bases: `torch.nn.modules.module.Module`

forward (*x*)

T_destination = **~T_destination**

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double() → T

Casts all floating point parameters and buffers to `double` datatype.

Returns: Module: self

dump_patches = False

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's *state_dict()* function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's *state_dict()* function. Default: True

Returns:

NamedTuple with *missing_keys* and *unexpected_keys* fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: *prefix* (str): prefix to prepend to all buffer names. *recurse* (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
```

(continues on next page)

(continued from previous page)

```

        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

class flood_forecast.transformer_xl.lower_upper_config.**MetaEmbedding** (*meta_vector_dim*,
out-put_dim,
predic-tor_number,
predic-tor_order)

Bases: torch.nn.modules.module.Module

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as *self*. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: *self*

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: *self*

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
```

(continues on next page)

(continued from previous page)

```
<class 'torch.Tensor'> (20L,)  
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu** () → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double** () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self**dump_patches = False****eval** () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self**extra_repr** () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self**forward** (*input) → None**half** () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self**load_state_dict** (*state_dict: Dict[str, torch.Tensor], strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: `prefix (str)`: prefix to prepend to all parameter names. `recurse (bool)`: if `True`, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if `True`, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook*: *Callable[[...], torch.utils.hooks.RemovableHandle]*, *None*) →

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name*: *str*, *param*: *torch.nn.parameter.Parameter*) → *None*

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad*: *bool = True*) → *T*

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: `self`

share_memory () → *T*

state_dict (*destination=None*, *prefix=""*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

```
to(device=None, dtype=None, non_blocking=False)
```

```
to(dtype, non_blocking=False)
```

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
```

(continues on next page)

(continued from previous page)

```
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self**type** (*dst_type: Union[torch.dtype, str]*) → TCasts all parameters and buffers to *dst_type*.**Arguments:** *dst_type* (type or string): the desired type**Returns:** Module: self**zero_grad** () → None

Sets gradients of all model parameters to zero.

Simple Multi Attention Head Model

```
class flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple (number_time_series:
                                                                    int,
                                                                    seq_len=10,
                                                                    out-
                                                                    put_seq_len=None,
                                                                    d_model=128,
                                                                    num_heads=8,
                                                                    fore-
                                                                    cast_length=None,
                                                                    dropout=0.1,
                                                                    sig-
                                                                    moid=False)
```

Bases: `torch.nn.modules.module.Module`

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

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to `double` datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to `float` datatype.

Returns: Module: self

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

Parameters `torch.Tensor (x)` – of shape (B, L, M)

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

half () → T

Casts all floating point parameters and buffers to `half` datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: `True`

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys

- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (memo: Optional[Set[Module]] = None, prefix: str = "")

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```

>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())

```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```

>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name*: str, *tensor*: torch.Tensor, *persistent*: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

`name` (string): name of the buffer. The buffer can be accessed from this module using the given name

`tensor` (Tensor): buffer to be registered. **`persistent` (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook*: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook*: Callable[[...], torch.utils.hooks.RemovableHandle], *hook*: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

CHAPTER 22

Basic Transformer

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

Bases: torch.nn.modules.module.Module

Full transformer model

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

basic_feature (x: torch.Tensor)

encode_sequence (x, src_mask=None)

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

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16 () → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double() → T

Casts all floating point parameters and buffers to `double` datatype.

Returns: Module: self

dump_patches = False

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to `half` datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor], strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict`

is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

NamedTuple with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if `True`, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix* (str): prefix to prepend to all parameter names. *recurse* (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook*: Callable[[...], torch.utils.hooks.RemovableHandle], *None*) → *torch.utils.hooks.RemovableHandle*
Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_parameter` (*name*: str, *param*: torch.nn.parameter.Parameter) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

`name` (string): name of the parameter. The parameter can be accessed from this module using the given name

`param` (Parameter): parameter to be added to the module.

`requires_grad_` (*requires_grad*: bool = True) → T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

`requires_grad` (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

`share_memory` () → T

`state_dict` (*destination*=None, *prefix*="", *keep_vars*=False)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

`dict`: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

to (device=None, dtype=None, non_blocking=False)

to (dtype, non_blocking=False)

to (tensor, non_blocking=False)

to (memory_format=torch.channels_last)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (True) or evaluation mode (False). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

```

class flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder (seq_length:
                                                                              int,
                                                                              out-
                                                                              put_seq_length:
                                                                              int,
                                                                              n_time_series:
                                                                              int,
                                                                              d_model=128,
                                                                              out-
                                                                              put_dim=1,
                                                                              n_layers_encode
                                                                              for-
                                                                              ward_dim=2048,
                                                                              dropout=0.1,
                                                                              use_mask=False,
                                                                              n_heads=8)

```

Bases: torch.nn.modules.module.Module

Uses a number of encoder layers with simple linear decoder layer

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

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

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as *self*. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: *self*

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16 () → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: *self*

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double() → T

Casts all floating point parameters and buffers to `double` datatype.

Returns: Module: self

dump_patches = False

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to `half` datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor], strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: `prefix (str)`: prefix to prepend to all parameter names. `recurse (bool)`: if `True`, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if `True`, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook*: *Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name*: *str*, *param*: *torch.nn.parameter.Parameter*) → *None*

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad*: *bool = True*) → *T*

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: `self`

share_memory () → *T*

state_dict (*destination=**None*, *prefix=**"*, *keep_vars=**False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

```
to(device=None, dtype=None, non_blocking=False)
to(dtype, non_blocking=False)
to(tensor, non_blocking=False)
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
```

(continues on next page)

(continued from previous page)

```
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self**type** (*dst_type: Union[torch.dtype, str]*) → TCasts all parameters and buffers to *dst_type*.**Arguments:** *dst_type* (type or string): the desired type**Returns:** Module: self**zero_grad** () → None

Sets gradients of all model parameters to zero.

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

Bases: torch.nn.modules.module.Module

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

Creates a basic positional encoding

T_destination = ~T_destination**add_module** (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule**Returns:** Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
```

(continues on next page)

(continued from previous page)

```

>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu**() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)

```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```

>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())

```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```

>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)

```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
 Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None
 Sets gradients of all model parameters to zero.

`flood_forecast.transformer_xl.transformer_basic.generate_square_subsequent_mask` (*sz: int*)
 → torch.Tensor

Generate a square mask for the sequence. The masked positions are filled with float('-inf'). Unmasked positions are filled with float(0.0).

`flood_forecast.transformer_xl.transformer_basic.greedy_decode` (*model, src: torch.Tensor, max_len: int, real_target: torch.Tensor, unsqueeze_dim=1, device='cpu'*)

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

CHAPTER 23

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,  
                                                                    dropouts:  
                                                                    float  
                                                                    =  
                                                                    0.0)
```

Bases: torch.nn.modules.module.Module

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)

T_destination = ~**T_destination**

add_module(*name*: str, *module*: torch.nn.modules.module.Module) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as *self*. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: *self*

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16 () → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: *self*

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu () → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (device: Union[int, torch.device, None] = None) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: True

Returns:**NamedTuple with missing_keys and unexpected_keys fields:**

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (memo: Optional[Set[Module]] = None, prefix: str = "")

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, grad_input and grad_output will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use torch.Tensor.register_hook() directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

`name (string)`: name of the buffer. The buffer can be accessed from this module using the given name

`tensor (Tensor)`: buffer to be registered. **`persistent (bool)`:** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook: Callable[[...], torch.utils.hooks.RemovableHandle]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

`to (tensor, non_blocking=False)`

`to (memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None
Sets gradients of all model parameters to zero.

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

Bases: torch.nn.modules.module.Module

forward (*input_: torch.FloatTensor*) → torch.FloatTensor

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if **True**, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double()** → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False**

eval() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if **True**, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. persistent (bool): whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (name: str, param: torch.nn.parameter.Parameter) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (requires_grad: bool = True) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

```
class flood_forecast.transformer_xl.transformer_xl.DecoderBlock (n_heads,
                                                                d_input,
                                                                d_head_inner,
                                                                d_ff_inner,
                                                                dropout,
                                                                dropouts=0.0)
```

Bases: torch.nn.modules.module.Module

forward (*input_*: torch.FloatTensor, *pos_embs*: torch.FloatTensor, *u*: torch.FloatTensor, *v*: torch.FloatTensor, *mask*=None, *mems*=None)

T_destination = ~**T_destination**

add_module (*name*: str, *module*: torch.nn.modules.module.Module) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn*: Callable[[Module], None]) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16 () → T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Returns: Module: self

buffers (*recurse*: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if **True**, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu () → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (device: Union[int, torch.device, None] = None) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to `double` datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to `half` datatype.

Returns: Module: self

load_state_dict (*state_dict*: Dict[str, torch.Tensor], *strict*: bool = True)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook*: `Callable[[...], None]`) $\rightarrow$ `torch.utils.hooks.RemovableHandle`
Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

`hook(module, input) -> None or modified input`

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_parameter` (*name*: `str`, *param*: `torch.nn.parameter.Parameter`) $\rightarrow$ `None`
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

`name (string)`: name of the parameter. The parameter can be accessed from this module using the given name

`param (Parameter)`: parameter to be added to the module.

`requires_grad_` (*requires_grad*: `bool = True`) $\rightarrow$ `T`
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

`requires_grad (bool)`: whether autograd should record operations on parameters in this module.
Default: `True`.

Returns: Module: `self`

`share_memory` () $\rightarrow$ `T`

`state_dict` (*destination*=`None`, *prefix*=`"`, *keep_vars*=`False`)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

`dict`: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

```
to(device=None, dtype=None, non_blocking=False)
```

```
to(dtype, non_blocking=False)
```

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (True) or evaluation mode (False). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

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

Bases: torch.nn.modules.module.Module

forward (*positions: torch.LongTensor*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```

>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self

buffers (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module

cpu() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self

cuda (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
    print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
    print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```

>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())

```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```

>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that

will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_buffer` (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

`register_forward_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

`register_forward_pre_hook` (*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad() → None
Sets gradients of all model parameters to zero.

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

Bases: torch.nn.modules.module.Module

forward (*input_: torch.LongTensor*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
```

(continues on next page)

(continued from previous page)

```

(0): Linear(in_features=2, out_features=2, bias=True)
(1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu**() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double**() → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False****eval**() → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict(*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (memo: Optional[Set[Module]] = None, prefix: str = "")

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. persistent (bool): whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (name: str, param: torch.nn.parameter.Parameter) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (requires_grad: bool = True) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

```

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,
                                                                    dropouth=0.0,
                                                                    seq_len:
                                                                    int      = 0,
                                                                    mem_len: int
                                                                    = 0)

```

Bases: torch.nn.modules.module.Module

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]

T_destination = ~T_destination

add_module (name: str, module: torch.nn.modules.module.Module) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (fn: Callable[[Module], None]) → T

Applies fn recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (Module -> None): function to be applied to each submodule

Returns: Module: self

Example:

```

>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],

```

(continues on next page)

(continued from previous page)

```

        [ 1.,  1.])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double()** → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False****eval()** → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (memo: Optional[Set[Module]] = None, prefix: str = "")

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if **True**, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. persistent (bool): whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (name: str, param: torch.nn.parameter.Parameter) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (requires_grad: bool = True) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module.
Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

CHAPTER 24

Basic GCP Utils

```
flood_forecast.gcp_integration.basic_utils.get_storage_client() →  
                                                                    google.cloud.storage.client.Client  
    Utility function to return a properly authenticated GCS storage client whether working in Colab, CircleCI, or  
    other environment.  
  
flood_forecast.gcp_integration.basic_utils.upload_file(bucket_name:      str,  
                                                         file_name:      str,  up-  
                                                         load_name:    str,  client:  
                                                         google.cloud.storage.client.Client)  
  
flood_forecast.gcp_integration.basic_utils.create_file_environ()
```


Train da

```
flood_forecast.da_rnn.train_da.da_rnn(train_data: flood_forecast.da_rnn.custom_types.TrainData,
                                         n_targs: int, encoder_hidden_size=64, de-
                                         coder_hidden_size=64, T=10, learning_rate=0.01,
                                         batch_size=128, param_output_path="",
                                         save_path: str = None) → Tuple[dict,
                                         flood_forecast.da_rnn.custom_types.DaRnnNet]
    n_targs: The number of target columns (not steps) T: The number timesteps in the window
flood_forecast.da_rnn.train_da.train(net: flood_forecast.da_rnn.custom_types.DaRnnNet,
                                       train_data: flood_forecast.da_rnn.custom_types.TrainData,
                                       t_cfg: flood_forecast.da_rnn.custom_types.TrainConfig,
                                       train_config="", n_epochs=10, save_plots=True,
                                       wandb=False, tensorboard=False)
flood_forecast.da_rnn.train_da.prep_train_data(batch_idx: numpy.ndarray, t_cfg:
                                                flood_forecast.da_rnn.custom_types.TrainConfig,
                                                train_data:
                                                flood_forecast.da_rnn.custom_types.TrainData)
                                                → Tuple
flood_forecast.da_rnn.train_da.adjust_learning_rate(net:
                                                       flood_forecast.da_rnn.custom_types.DaRnnNet,
                                                       n_iter: int) → None
flood_forecast.da_rnn.train_da.train_iteration(t_net: flood_forecast.da_rnn.custom_types.DaRnnNet,
                                                  loss_func: Callable, X, y_history,
                                                  y_target)
flood_forecast.da_rnn.train_da.predict(t_net: flood_forecast.da_rnn.custom_types.DaRnnNet,
                                         t_dat: flood_forecast.da_rnn.custom_types.TrainData,
                                         train_size: int, batch_size: int, T: int,
                                         on_train=False)
```


CHAPTER 26

Utils

```
flood_forecast.da_rnn.utils.setup_log(tag='VOC_TOPICS')  
flood_forecast.da_rnn.utils.save_or_show_plot(file_nm: str, save: bool, save_path="")  
flood_forecast.da_rnn.utils.numpy_to_tvar(x)
```



```
class flood_forecast.da_rnn.custom_types.TrainConfig(T, train_size, batch_size,  
                                                    loss_func)
```

Bases: `tuple`

Create new instance of TrainConfig(T, train_size, batch_size, loss_func)

T

Alias for field number 0

train_size

Alias for field number 1

batch_size

Alias for field number 2

loss_func

Alias for field number 3

count ()

Return number of occurrences of value.

index ()

Return first index of value.

Raises ValueError if the value is not present.

```
class flood_forecast.da_rnn.custom_types.TrainData(feats, targs)
```

Bases: `tuple`

Create new instance of TrainData(feats, targs)

feats

Alias for field number 0

targs

Alias for field number 1

count ()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

class flood_forecast.da_rnn.custom_types.**DaRnnNet** (*encoder, decoder, enc_opt, dec_opt*)

Bases: `tuple`

Create new instance of DaRnnNet(encoder, decoder, enc_opt, dec_opt)

count()

Return number of occurrences of value.

dec_opt

Alias for field number 3

decoder

Alias for field number 1

enc_opt

Alias for field number 2

encoder

Alias for field number 0

index()

Return first index of value.

Raises ValueError if the value is not present.

```
class flood_forecast.da_rnn.model.DARNN(input_size: int, hidden_size_encoder: int, T: int,
                                         decoder_hidden_size: int, out_feats=1)
```

Bases: torch.nn.modules.module.Module

input size: number of underlying factors (81) T: number of time steps (10) hidden_size: dimension of the hidden state

forward (x: torch.Tensor, y_history: torch.Tensor)
will implement

T_destination = ~T_destination

add_module (name: str, module: torch.nn.modules.module.Module) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (fn: Callable[[Module], None]) → T

Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
```

(continues on next page)

(continued from previous page)

```

>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu**() → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)

```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```

>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())

```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```

>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)

```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to `dst_type`.

Arguments: `dst_type` (type or string): the desired type

Returns: Module: `self`

zero_grad () → None

Sets gradients of all model parameters to zero.

`flood_forecast.da_rnn.modules.init_hidden(x, hidden_size: int)`

Train the initial value of the hidden state: <https://r2rt.com/non-zero-initial-states-for-recurrent-neural-networks.html>

class `flood_forecast.da_rnn.modules.Encoder(input_size: int, hidden_size: int, T: int)`

Bases: `torch.nn.modules.module.Module`

input_size: number of underlying factors (81) T: number of time steps (10) hidden_size: dimension of the hidden state

forward (`input_data: torch.Tensor`)

T_destination = ~T_destination

add_module (`name: str, module: torch.nn.modules.module.Module`) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (`fn: Callable[[Module], None]`) → T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: `fn (Module -> None)`: function to be applied to each submodule

Returns: Module: `self`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
```

(continues on next page)

(continued from previous page)

```

>>> print(m)
>>> if type(m) == nn.Linear:
>>>     m.weight.fill_(1.0)
>>>     print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)

```

bfloat16() → T

Casts all floating point parameters and buffers to bfloat16 datatype.

Returns: Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```

>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

double () → T

Casts all floating point parameters and buffers to double datatype.

Returns: Module: self

dump_patches = False

eval () → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

Returns: Module: self

extra_repr () → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float () → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half () → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (state_dict: Dict[str, torch.Tensor], strict: bool = True)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict ()` function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict ()` function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules () → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)

```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```

>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())

```

named_children () → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```

>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)

```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: *prefix (str):* prefix to prepend to all parameter names. *recurse (bool):* if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name: str, param: torch.nn.parameter.Parameter*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad: bool = True*) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (**args, **kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (torch.device): the desired device of the parameters and buffers in this module

dtype (torch.dtype): the desired floating point type of the floating point parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (torch.memory_format): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (mode: bool = True) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: True.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T
 Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None
 Sets gradients of all model parameters to zero.

class `flood_forecast.da_rnn.modules.Decoder` (*encoder_hidden_size: int, decoder_hidden_size: int, T: int, out_feats=1*)

Bases: `torch.nn.modules.module.Module`

forward (*input_encoded, y_history*)

T_destination = ~T_destination

add_module (*name: str, module: torch.nn.modules.module.Module*) → None
 Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

apply (*fn: Callable[[Module], None]*) → T
 Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

Args: *fn* (Module → None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
(1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → TCasts all floating point parameters and buffers to `bfloat16` datatype.**Returns:** Module: self**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Args:**recurse (bool):** if **True**, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields:** torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields: Module: a child module**cpu()** → T

Moves all model parameters and buffers to the CPU.

Returns: Module: self**cuda** (*device: Union[int, torch.device, None] = None*) → T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Arguments:**device (int, optional):** if specified, all parameters will be copied to that device**Returns:** Module: self**double()** → TCasts all floating point parameters and buffers to `double` datatype.**Returns:** Module: self**dump_patches = False****eval()** → T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.**Returns:** Module: self

extra_repr() → str

Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → T

Casts all floating point parameters and buffers to float datatype.

Returns: Module: self

half() → T

Casts all floating point parameters and buffers to half datatype.

Returns: Module: self

load_state_dict (*state_dict: Dict[str, torch.Tensor]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If *strict* is True, then the keys of *state_dict* must exactly match the keys returned by this module's *state_dict()* function.

Arguments:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's *state_dict()* function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo: Optional[Set[Module]] = None, prefix: str = ""*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters (*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters (*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook (*hook: Callable[[Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

Warning: The current implementation will not have the presented behavior for complex Module that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such Module, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_buffer (*name: str, tensor: torch.Tensor, persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor): buffer to be registered. **persistent (bool):** whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook (*hook*: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_forward_pre_hook (*hook*: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

register_parameter (*name*: str, *param*: torch.nn.parameter.Parameter) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter): parameter to be added to the module.

requires_grad_ (*requires_grad*: bool = True) → T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: True.

Returns: Module: self

share_memory () → T

state_dict (*destination=None, prefix="", keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to (*args, **kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

to (*device=None, dtype=None, non_blocking=False*)

to (*dtype, non_blocking=False*)

to (*tensor, non_blocking=False*)

to (*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype`s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with `dtypes` unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point type of the floating point parameters and buffers in this module

tensor (`torch.Tensor`): Tensor whose `dtype` and `device` are the desired `dtype` and `device` for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
```

(continues on next page)

(continued from previous page)

```

        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

```

train (*mode: bool = True*) → T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Args:

mode (bool): whether to set training mode (**True**) or evaluation mode (**False**). Default: **True**.

Returns: Module: self

type (*dst_type: Union[torch.dtype, str]*) → T

Casts all parameters and buffers to *dst_type*.

Arguments: *dst_type* (type or string): the desired type

Returns: Module: self

zero_grad () → None

Sets gradients of all model parameters to zero.

CHAPTER 30

Indices and tables

- `genindex`
- `modindex`
- `search`

f

`flood_forecast`, ??
`flood_forecast.custom`, 35
`flood_forecast.custom.custom_opt`, 37
`flood_forecast.da_rnn`, 185
`flood_forecast.da_rnn.custom_types`, 191
`flood_forecast.da_rnn.model`, 193
`flood_forecast.da_rnn.modules`, 203
`flood_forecast.da_rnn.train_da`, 187
`flood_forecast.da_rnn.utils`, 189
`flood_forecast.evaluator`, 3
`flood_forecast.gcp_integration`, 183
`flood_forecast.gcp_integration.basic_utils`, 185
`flood_forecast.long_train`, 7
`flood_forecast.model_dict_function`, 9
`flood_forecast.pre_dict`, 11
`flood_forecast.preprocessing`, 17
`flood_forecast.preprocessing.buil_dataset`, 21
`flood_forecast.preprocessing.closest_station`, 23
`flood_forecast.preprocessing.data_converter`, 25
`flood_forecast.preprocessing.interpolate_preprocess`, 19
`flood_forecast.preprocessing.preprocess_da_rnn`, 27
`flood_forecast.preprocessing.preprocess_metadata`, 29
`flood_forecast.preprocessing.process_usgs`, 31
`flood_forecast.preprocessing.pytorch_loaders`, 33
`flood_forecast.preprocessing.temporal_feats`, 35
`flood_forecast.pytorch_training`, 13
`flood_forecast.time_model`, 15
`flood_forecast.trainer`, 17
`flood_forecast.transformer_xl`, 68
`flood_forecast.transformer_xl.dummy_torch`, 69
`flood_forecast.transformer_xl.lower_upper_config`, 77
`flood_forecast.transformer_xl.multi_head_base`, 101
`flood_forecast.transformer_xl.transformer_basic`, 111
`flood_forecast.transformer_xl.transformer_xl`, 137
`flood_forecast.utils`, 1

A

`add_module()` (*flood_forecast.custom.custom_opt.GaussianLoss* *method*), 168
 method), 52
`add_module()` (*flood_forecast.custom.custom_opt.MAPELoss* *method*), 176
 method), 45
`add_module()` (*flood_forecast.custom.custom_opt.QuantileLoss* *method*), 60
 method), 60
`add_module()` (*flood_forecast.custom.custom_opt.RMSELoss* *method*), 37
 method), 37
`add_module()` (*flood_forecast.da_rnn.model.DARNN* *method*), 193
 method), 193
`add_module()` (*flood_forecast.da_rnn.modules.Decoder* *method*), 211
 method), 211
`add_module()` (*flood_forecast.da_rnn.modules.Encoder* *method*), 203
 method), 203
`add_module()` (*flood_forecast.transformer_xl.dummy_torch.DummyTorchModel* *method*), 69
 method), 69
`add_module()` (*flood_forecast.transformer_xl.lower_upper_config.AR* *method*), 85
 method), 85
`add_module()` (*flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding* *method*), 92
 method), 92
`add_module()` (*flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward* *method*), 77
 method), 77
`add_module()` (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple* *method*), 101
 method), 101
`add_module()` (*flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder* *method*), 119
 method), 119
`add_module()` (*flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding* *method*), 127
 method), 127
`add_module()` (*flood_forecast.transformer_xl.transformer_basic.SimpleTransformer* *method*), 112
 method), 112
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock* *method*), 153
 method), 153
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention* *method*), 137
 method), 137
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding* *method*), 160
 method), 160
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFF* *method*), 145
 method), 145
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.StandardV* *method*), 168
 method), 168
`add_module()` (*flood_forecast.transformer_xl.transformer_xl.Transform* *method*), 176
 method), 176
`add_param_group()`
 (*flood_forecast.custom.custom_opt.BertAdam* *method*), 68
 method), 68
`adjust_learning_rate()` (in *module*
 flood_forecast.da_rnn.train_da), 187
`apply()` (*flood_forecast.custom.custom_opt.GaussianLoss* *method*), 53
 method), 53
`apply()` (*flood_forecast.custom.custom_opt.MAPELoss* *method*), 45
 method), 45
`apply()` (*flood_forecast.custom.custom_opt.QuantileLoss* *method*), 60
 method), 60
`apply()` (*flood_forecast.custom.custom_opt.RMSELoss* *method*), 37
 method), 37
`apply()` (*flood_forecast.da_rnn.model.DARNN* *method*), 193
 method), 193
`apply()` (*flood_forecast.da_rnn.modules.Decoder* *method*), 211
 method), 211
`apply()` (*flood_forecast.da_rnn.modules.Encoder* *method*), 203
 method), 203
`apply()` (*flood_forecast.transformer_xl.dummy_torch.DummyTorchModel* *method*), 69
 method), 69
`apply()` (*flood_forecast.transformer_xl.lower_upper_config.AR* *method*), 85
 method), 85
`apply()` (*flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding* *method*), 93
 method), 93
`apply()` (*flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward* *method*), 77
 method), 77
`apply()` (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple* *method*), 101
 method), 101
`apply()` (*flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder* *method*), 120
 method), 120
`apply()` (*flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding* *method*), 127
 method), 127
`apply()` (*flood_forecast.transformer_xl.transformer_basic.SimpleTransformer* *method*), 112
 method), 112
`apply()` (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock* *method*), 153
 method), 153
`apply()` (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention* *method*), 137
 method), 137
`apply()` (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding* *method*), 160
 method), 160
`apply()` (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFF* *method*), 145
 method), 145
`apply()` (*flood_forecast.transformer_xl.transformer_xl.StandardV* *method*), 168
 method), 168
`apply()` (*flood_forecast.transformer_xl.transformer_xl.Transform* *method*), 176
 method), 176

method), 153
 apply () (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention), 153
 method), 138
 apply () (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding), 138
 method), 160
 apply () (flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward), 161
 method), 145
 apply () (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding), 169
 method), 168
 apply () (flood_forecast.transformer_xl.transformer_xl.TransformerXL), 169
 method), 176
 AR (class in flood_forecast.transformer_xl.lower_upper_config), method), 177
 85
 AR.to () (in module flood_forecast.transformer_xl.lower_upper_config), method), 53
 91
B
 basic_feature () (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer), 38
 method), 111
 batch_size (flood_forecast.da_rnn.custom_types.TrainConfig attribute), 191
 BertAdam (class in flood_forecast.custom.custom_opt), buffers () (flood_forecast.da_rnn.modules.Decoder), 67
 method), 212
 bfloat16 () (flood_forecast.custom.custom_opt.GaussianLoss), buffers () (flood_forecast.da_rnn.modules.Encoder), 53
 method), 204
 bfloat16 () (flood_forecast.custom.custom_opt.MAPELoss), buffers () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModule), 46
 method), 70
 bfloat16 () (flood_forecast.custom.custom_opt.QuantileLoss), buffers () (flood_forecast.transformer_xl.lower_upper_config.AR), 61
 method), 86
 bfloat16 () (flood_forecast.custom.custom_opt.RMSELoss), buffers () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding), 38
 method), 93
 bfloat16 () (flood_forecast.da_rnn.model.DARNN), buffers () (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward), 194
 method), 78
 bfloat16 () (flood_forecast.da_rnn.modules.Decoder), buffers () (flood_forecast.transformer_xl.multi_head_base.MultiAttnHead), 212
 method), 102
 bfloat16 () (flood_forecast.da_rnn.modules.Encoder), buffers () (flood_forecast.transformer_xl.transformer_basic.CustomTransformer), 204
 method), 120
 bfloat16 () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModule), buffers () (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer), 70
 method), 128
 bfloat16 () (flood_forecast.transformer_xl.lower_upper_config.AR), buffers () (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer), 85
 method), 112
 bfloat16 () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding), buffers () (flood_forecast.transformer_xl.transformer_xl.DecoderBlock), 93
 method), 153
 bfloat16 () (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward), buffers () (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention), 78
 method), 138
 bfloat16 () (flood_forecast.transformer_xl.multi_head_base.MultiAttnHead), buffers () (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding), 102
 method), 161
 bfloat16 () (flood_forecast.transformer_xl.transformer_basic.CustomTransformer), buffers () (flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward), 120
 method), 146
 bfloat16 () (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding), buffers () (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding), 128
 method), 169
 bfloat16 () (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer), buffers () (flood_forecast.transformer_xl.transformer_xl.TransformerXL), 112
 method), 177

build_weather_csv() (in module flood_forecast.preprocessing.buil_dataset), 21
 C
 check_loss() (flood_forecast.utils.EarlyStopper method), 1
 children() (flood_forecast.custom.custom_opt.GaussianLoss method), 53
 children() (flood_forecast.custom.custom_opt.MAPELoss method), 46
 children() (flood_forecast.custom.custom_opt.QuantileLoss method), 61
 children() (flood_forecast.custom.custom_opt.RMSELoss method), 38
 children() (flood_forecast.da_rnn.model.DARNN method), 194
 children() (flood_forecast.da_rnn.modules.Decoder method), 212
 children() (flood_forecast.da_rnn.modules.Encoder method), 204
 children() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 70
 children() (flood_forecast.transformer_xl.lower_upper_config.AR method), 86
 children() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94
 children() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward method), 78
 children() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple method), 102
 children() (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder method), 121
 children() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding method), 128
 children() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalFeeding method), 113
 children() (flood_forecast.transformer_xl.transformer_decoder.DecoderBlock method), 154
 children() (flood_forecast.transformer_xl.transformer_decoder.MultiHeadAttention method), 138
 children() (flood_forecast.transformer_xl.transformer_decoder.PositionalEmbedding method), 161
 children() (flood_forecast.transformer_xl.transformer_decoder.PositionwiseFF method), 146
 children() (flood_forecast.transformer_xl.transformer_decoder.StandardWordEmbeddings method), 169
 children() (flood_forecast.transformer_xl.transformer_decoder.TransformerXL method), 177
 combine_data() (in module flood_forecast.preprocessing.buil_dataset), 21
 compute_validation() (in module flood_forecast.pytorch_training), 13
 convert_history_batches() (flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader method), 34
 convert_real_batches() (flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader method), 34
 convert_temp() (in module flood_forecast.preprocessing.closest_station), 23
 count() (flood_forecast.da_rnn.custom_types.DaRnnNet method), 192
 count() (flood_forecast.da_rnn.custom_types.TrainConfig method), 191
 count() (flood_forecast.da_rnn.custom_types.TrainData method), 191
 cpu() (flood_forecast.custom.custom_opt.GaussianLoss method), 54
 cpu() (flood_forecast.custom.custom_opt.MAPELoss method), 46
 cpu() (flood_forecast.custom.custom_opt.QuantileLoss method), 61
 cpu() (flood_forecast.custom.custom_opt.RMSELoss method), 38
 cpu() (flood_forecast.da_rnn.model.DARNN method), 194
 cpu() (flood_forecast.da_rnn.modules.Decoder method), 212
 cpu() (flood_forecast.da_rnn.modules.Encoder method), 204
 cpu() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 70
 cpu() (flood_forecast.transformer_xl.lower_upper_config.AR method), 86
 cpu() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94
 cpu() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward method), 78
 cpu() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple method), 102
 cpu() (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder method), 121
 cpu() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding method), 128
 cpu() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalFeeding method), 113
 cpu() (flood_forecast.transformer_xl.transformer_decoder.DecoderBlock method), 154
 cpu() (flood_forecast.transformer_xl.transformer_decoder.MultiHeadAttention method), 138
 cpu() (flood_forecast.transformer_xl.transformer_decoder.PositionalEmbedding method), 161
 cpu() (flood_forecast.transformer_xl.transformer_decoder.PositionwiseFF method), 146
 cpu() (flood_forecast.transformer_xl.transformer_decoder.StandardWordEmbeddings method), 169
 cpu() (flood_forecast.transformer_xl.transformer_decoder.TransformerXL method), 177

method), 169

cpu() (flood_forecast.transformer_xl.transformer_xl.TransformerXLmethod), 161

method), 177

create_csv() (in module flood_forecast.preprocessing.process_usgs), 31

create_file_envIRON() (in module flood_forecast.gcp_integration.basic_utils), 185

create_usgs() (in module flood_forecast.preprocessing.buil_dataset), 21

create_visited() (in module flood_forecast.preprocessing.buil_dataset), 21

CSVDataLoader (class in flood_forecast.preprocessing.pytorch_loaders), 33

CSVTestLoader (class in flood_forecast.preprocessing.pytorch_loaders), 34

cuda() (flood_forecast.custom.custom_opt.GaussianLoss method), 54

cuda() (flood_forecast.custom.custom_opt.MAPELoss method), 46

cuda() (flood_forecast.custom.custom_opt.QuantileLoss method), 61

cuda() (flood_forecast.custom.custom_opt.RMSELoss method), 38

cuda() (flood_forecast.da_rnn.model.DARNN method), 194

cuda() (flood_forecast.da_rnn.modules.Decoder method), 212

cuda() (flood_forecast.da_rnn.modules.Encoder method), 204

cuda() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 70

cuda() (flood_forecast.transformer_xl.lower_upper_config.AR method), 86

cuda() (flood_forecast.transformer_xl.lower_upper_config.MeanEmbedding method), 94

cuda() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF method), 78

cuda() (flood_forecast.transformer_xl.multi_head_base.MultiHeadSimple method), 102

cuda() (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder method), 121

cuda() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEmbedding method), 128

cuda() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer method), 113

cuda() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock method), 154

cuda() (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention method), 139

cuda() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding method), 169

cuda() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF method), 146

cuda() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding method), 169

cuda() (flood_forecast.transformer_xl.transformer_xl.TransformerXL method), 177

CustomTransformerDecoder (class in flood_forecast.transformer_xl.transformer_basic), 119

CustomTransformerDecoder.to() (in module flood_forecast.transformer_xl.transformer_basic), 126

D

da_rnn() (in module flood_forecast.da_rnn.train_da), 187

DARNN (class in flood_forecast.da_rnn.model), 193

DARNN.to() (in module flood_forecast.da_rnn.model), 199

DaRnnNet (class in flood_forecast.da_rnn.custom_types), 192

dec_opt (flood_forecast.da_rnn.custom_types.DaRnnNet attribute), 192

decode_seq() (flood_forecast.transformer_xl.transformer_basic.Simple method), 111

Decoder (class in flood_forecast.da_rnn.modules), 211

decoder (flood_forecast.da_rnn.custom_types.DaRnnNet attribute), 192

Decoder.to() (in module flood_forecast.da_rnn.modules), 217

DecoderBlock (class in flood_forecast.transformer_xl.transformer_xl), 154

DecoderBlock.to() (in module flood_forecast.transformer_xl.transformer_xl), 159

MetaEmbedding (in module flood_forecast.preprocessing.process_usgs), 31

PositionwiseFF (flood_forecast.custom.custom_opt.GaussianLoss method), 54

SimplePositionalEmbedding (flood_forecast.custom.custom_opt.MAPELoss method), 46

SimpleTransformer (flood_forecast.custom.custom_opt.QuantileLoss method), 61

SimpleTransformerEnforcing (flood_forecast.custom.custom_opt.RMSELoss method), 39

SimpleTransformer(flood_forecast.da_rnn.model.DARNN method), 195

SimpleTransformerBlock (flood_forecast.da_rnn.modules.Decoder method), 212

SimpleTransformer(flood_forecast.da_rnn.modules.Encoder method), 205

double () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF method), 70 attribute), 79

double () (flood_forecast.transformer_xl.lower_upper_config.AR (flood_forecast.transformer_xl.multi_head_base.MultiHeadAttention method), 86 attribute), 103

double () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding (flood_forecast.transformer_xl.transformer_basic.CustomTransformer method), 94 attribute), 121

double () (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF (flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF method), 79 attribute), 129

double () (flood_forecast.transformer_xl.multi_head_base.MultiHeadAttention (flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF method), 103 attribute), 113

double () (flood_forecast.transformer_xl.transformer_basic.CustomTransformer (flood_forecast.transformer_xl.transformer_xl.DecoderBlock method), 121 attribute), 154

double () (flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention method), 129 attribute), 139

double () (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF method), 113 attribute), 162

double () (flood_forecast.transformer_xl.transformer_xl.DecoderBlock (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF method), 154 attribute), 146

double () (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention (flood_forecast.transformer_xl.transformer_xl.StandardVocabulary method), 139 attribute), 169

double () (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF (flood_forecast.transformer_xl.transformer_xl.Transformers method), 162 attribute), 177

double () (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF (flood_forecast.transformer_xl.transformer_xl.Transformers method), 146 attribute), 177

double () (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding (flood_forecast.transformer_xl.transformer_xl.StandardVocabulary method), 169 attribute), 1

double () (flood_forecast.transformer_xl.transformer_xl.Transformers (flood_forecast.transformer_xl.transformer_xl.Transformers method), 177 attribute), 192

DummyTorchModel (class in flood_forecast.transformer_xl.dummy_torch), 69 (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer method), 111

DummyTorchModel.to () (in module flood_forecast.transformer_xl.dummy_torch), 75 Encoder (class in flood_forecast.da_rnn.modules), 203 encoder (flood_forecast.da_rnn.custom_types.DaRnnNet attribute), 192 Encoder.to () (in module flood_forecast.da_rnn.modules), 209

dump_patches (flood_forecast.custom.custom_opt.GaussianLoss (flood_forecast.da_rnn.modules), 209 attribute), 54 eval () (flood_forecast.custom.custom_opt.GaussianLoss method), 54

dump_patches (flood_forecast.custom.custom_opt.MAPELoss (flood_forecast.da_rnn.modules), 209 attribute), 46 eval () (flood_forecast.custom.custom_opt.MAPELoss method), 46

dump_patches (flood_forecast.custom.custom_opt.QuantileLoss (flood_forecast.da_rnn.modules), 209 attribute), 61 eval () (flood_forecast.custom.custom_opt.QuantileLoss method), 61

dump_patches (flood_forecast.custom.custom_opt.RMSELoss (flood_forecast.da_rnn.modules), 209 attribute), 39 eval () (flood_forecast.custom.custom_opt.RMSELoss method), 39

dump_patches (flood_forecast.da_rnn.model.DARNN (flood_forecast.da_rnn.model.DARNN method), 195 attribute), 195 eval () (flood_forecast.da_rnn.model.DARNN method), 195

dump_patches (flood_forecast.da_rnn.modules.Decoder (flood_forecast.da_rnn.modules.Decoder method), 212 attribute), 212 eval () (flood_forecast.da_rnn.modules.Decoder method), 212

dump_patches (flood_forecast.da_rnn.modules.Encoder (flood_forecast.da_rnn.modules.Encoder method), 205 attribute), 205 eval () (flood_forecast.da_rnn.modules.Encoder method), 205

dump_patches (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 71 attribute), 71 eval () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 71

dump_patches (flood_forecast.transformer_xl.lower_upper_config.AR (flood_forecast.transformer_xl.lower_upper_config.AR method), 86 attribute), 86 eval () (flood_forecast.transformer_xl.lower_upper_config.AR method), 86

dump_patches (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94 attribute), 94 eval () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94

eval () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding (flood_forecast.transformer_xl.transformer_basic.Simple
 method), 94 method), 113
 eval () (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward (flood_forecast.transformer_xl.transformer_xl.DecoderB
 method), 79 method), 154
 eval () (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple (flood_forecast.transformer_xl.transformer_xl.MultiHead
 method), 103 method), 139
 eval () (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder (flood_forecast.transformer_xl.transformer_xl.Positional
 method), 121 method), 162
 eval () (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding (flood_forecast.transformer_xl.transformer_xl.Positionw
 method), 129 method), 147
 eval () (flood_forecast.transformer_xl.transformer_basic.StandardWordEmbedding (flood_forecast.transformer_xl.transformer_xl.StandardV
 method), 113 method), 170
 eval () (flood_forecast.transformer_xl.transformer_xl.DecoderBlock repr () (flood_forecast.transformer_xl.transformer_xl.Transform
 method), 154 method), 178
 eval () (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention
 method), 139
 eval () (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding (flood_forecast.da_rnn.custom_types.TrainData
 method), 162 attribute), 191
 eval () (flood_forecast.transformer_xl.transformer_xl.PositionwiseFE (in module
 method), 146 flood_forecast.preprocessing.interpolate_preprocess),
 eval () (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding
 method), 169 flatten_list_function () (in module
 eval () (flood_forecast.transformer_xl.transformer_xl.TransformerXL (flood_forecast.utils), 1
 method), 177 float () (flood_forecast.custom.custom_opt.GaussianLoss
 evaluate_model () (in module method), 54
 flood_forecast.evaluator), 3 float () (flood_forecast.custom.custom_opt.MAPELoss
 extra_repr () (flood_forecast.custom.custom_opt.GaussianLoss method), 47
 method), 54 float () (flood_forecast.custom.custom_opt.QuantileLoss
 extra_repr () (flood_forecast.custom.custom_opt.MAPELoss method), 62
 method), 46 float () (flood_forecast.custom.custom_opt.RMSELoss
 extra_repr () (flood_forecast.custom.custom_opt.QuantileLoss method), 39
 method), 62 float () (flood_forecast.da_rnn.model.DARNN
 extra_repr () (flood_forecast.custom.custom_opt.RMSELoss method), 195
 method), 39 float () (flood_forecast.da_rnn.modules.Decoder
 extra_repr () (flood_forecast.da_rnn.model.DARNN method), 213
 method), 195 float () (flood_forecast.da_rnn.modules.Encoder
 extra_repr () (flood_forecast.da_rnn.modules.Decoder method), 205
 method), 212 float () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel
 extra_repr () (flood_forecast.da_rnn.modules.Encoder method), 71
 method), 205 float () (flood_forecast.transformer_xl.lower_upper_config.AR
 extra_repr () (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 86
 method), 71 float () (flood_forecast.transformer_xl.lower_upper_config.Positionwise
 extra_repr () (flood_forecast.transformer_xl.lower_upper_config.AR method), 94
 method), 86 float () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding
 extra_repr () (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94
 method), 94 float () (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadS
 extra_repr () (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward method), 79
 method), 79 float () (flood_forecast.transformer_xl.transformer_basic.CustomTransf
 extra_repr () (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple method), 103
 method), 103 float () (flood_forecast.transformer_xl.transformer_basic.SimplePosition
 extra_repr () (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder method), 121
 method), 121 float () (flood_forecast.transformer_xl.transformer_basic.SimpleTransfo
 extra_repr () (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding method), 129
 method), 129

float () (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock*
 method), 154
 float () (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention*
 method), 139
 float () (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding*
 method), 162
 float () (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFF*
 method), 147
 float () (*flood_forecast.transformer_xl.transformer_xl.StandardizedWordEmbedding*
 method), 170
 float () (*flood_forecast.transformer_xl.transformer_xl.TransformXL*
 method), 178
 flood_forecast (*module*), 1
 flood_forecast.custom (*module*), 35
 flood_forecast.custom.custom_opt (*mod-
 ule*), 37
 flood_forecast.da_rnn (*module*), 185
 flood_forecast.da_rnn.custom_types (*mod-
 ule*), 191
 flood_forecast.da_rnn.model (*module*), 193
 flood_forecast.da_rnn.modules (*module*),
 203
 flood_forecast.da_rnn.train_da (*module*),
 187
 flood_forecast.da_rnn.utils (*module*), 189
 flood_forecast.evaluator (*module*), 3
 flood_forecast.gcp_integration (*module*),
 183
 flood_forecast.gcp_integration.basic_utils
 (*module*), 185
 flood_forecast.long_train (*module*), 7
 flood_forecast.model_dict_function (*mod-
 ule*), 9
 flood_forecast.pre_dict (*module*), 11
 flood_forecast.preprocessing (*module*), 17
 flood_forecast.preprocessing.buil_datase
 (*module*), 21
 flood_forecast.preprocessing.closest_sta
 (*module*), 23
 flood_forecast.preprocessing.data_conver
 (*module*), 25
 flood_forecast.preprocessing.interpolate
 (*module*), 19
 flood_forecast.preprocessing.preprocess_da
 (*module*), 27
 flood_forecast.preprocessing.preprocess_me
 (*module*), 29
 flood_forecast.preprocessing.process_usg
 (*module*), 31
 flood_forecast.preprocessing.pytorch_load
 (*module*), 33
 flood_forecast.preprocessing.temporal_fea
 (*module*), 35
 flood_forecast.pytorch_training (*module*),

flood_forecast.time_model (*module*), 15
 flood_forecast.trainer (*module*), 17
 flood_forecast.transformer_xl (*module*), 68
 flood_forecast.transformer_xl.dummy_torch
 (*module*), 69
 flood_forecast.transformer_xl.lower_upper_config
 (*module*), 77
 flood_forecast.transformer_xl.multi_head_base
 (*module*), 101
 flood_forecast.transformer_xl.transformer_basic
 (*module*), 111
 flood_forecast.transformer_xl.transformer_xl
 (*module*), 137
 flood_forecast.utils (*module*), 1
 format_data () (in *module*
 flood_forecast.preprocessing.preprocess_da_rnn),
 27
 format_dt () (in *module*
 flood_forecast.preprocessing.closest_station),
 23
 forward () (*flood_forecast.custom.custom_opt.GaussianLoss*
 method), 52
 forward () (*flood_forecast.custom.custom_opt.MAPELoss*
 method), 45
 forward () (*flood_forecast.custom.custom_opt.QuantileLoss*
 method), 60
 forward () (*flood_forecast.custom.custom_opt.RMSELoss*
 method), 37
 forward () (*flood_forecast.da_rnn.model.DARNN*
 method), 193
 forward () (*flood_forecast.da_rnn.modules.Decoder*
 method), 211
 forward () (*flood_forecast.da_rnn.modules.Encoder*
 method), 203
 forward () (*flood_forecast.transformer_xl.dummy_torch.DummyTorchMo*
 method), 69
 forward () (*flood_forecast.transformer_xl.lower_upper_config.AR*
 method), 85
 forward () (*flood_forecast.transformer_xl.lower_upper_config.MetaEmb*
 method), 94
 forward () (*flood_forecast.transformer_xl.lower_upper_config.Positionw*
 method), 77
 forward () (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHe*
 method), 103
 forward () (*flood_forecast.transformer_xl.transformer_basic.CustomTra*
 method), 119
 forward () (*flood_forecast.transformer_xl.transformer_basic.SimplePosi*
 method), 127
 forward () (*flood_forecast.transformer_xl.transformer_basic.SimpleTran*
 method), 111
 forward () (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock*
 method), 152
 forward () (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAtte*

method), 137

forward() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF (in module flood_forecast.preprocessing.process_usgs), 31 method), 160

forward() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF (in module flood_forecast.evaluator), 3 method), 145

forward() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding (in module flood_forecast.preprocessing.closest_station), method), 168

forward() (flood_forecast.transformer_xl.transformer_xl.TransformersXL (in module flood_forecast.preprocessing.temporal_feats), method), 176

G

GaussianLoss (class in module flood_forecast.custom.custom_opt), 52

GaussianLoss.to() (in module flood_forecast.custom.custom_opt), 58, 59

generate_decoded_predictions() (in module flood_forecast.evaluator), 4

generate_prediction_samples() (in module flood_forecast.evaluator), 4

generate_predictions() (in module flood_forecast.evaluator), 4

generate_predictions_non_decoded() (in module flood_forecast.evaluator), 4

generate_square_subsequent_mask() (in module flood_forecast.model_dict_function), 9

generate_square_subsequent_mask() (in module flood_forecast.transformer_xl.transformer_basic), 135

get_closest_gage() (in module flood_forecast.preprocessing.closest_station), 23

get_day() (in module flood_forecast.preprocessing.temporal_feats), 35

get_eco_netset() (in module flood_forecast.preprocessing.buil_dataset), 21

get_from_start_date() (flood_forecast.preprocessing.pytorch_loaders.CSVTestLoader method), 34

get_hour() (in module flood_forecast.preprocessing.temporal_feats), 35

get_lr() (flood_forecast.custom.custom_opt.BertAdam method), 68

get_model_r2_score() (in module flood_forecast.evaluator), 3

get_month() (in module flood_forecast.preprocessing.temporal_feats), 35

get_r2_value() (in module flood_forecast.evaluator), 3

get_storage_client() (in module flood_forecast.gcp_integration.basic_utils), 185

H

half() (flood_forecast.custom.custom_opt.GaussianLoss method), 54

half() (flood_forecast.custom.custom_opt.MAPELoss method), 47

half() (flood_forecast.custom.custom_opt.QuantileLoss method), 62

half() (flood_forecast.custom.custom_opt.RMSELoss method), 39

half() (flood_forecast.da_rnn.model.DARNN method), 195

half() (flood_forecast.da_rnn.modules.Decoder method), 213

half() (flood_forecast.da_rnn.modules.Encoder method), 205

half() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 71

half() (flood_forecast.transformer_xl.lower_upper_config.AR method), 87

half() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 94

half() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF method), 79

half() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple method), 103

half() (flood_forecast.transformer_xl.transformer_basic.CustomTransformer method), 121

half() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding method), 129

half() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer method), 113

half() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock method), 154

half() (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention method), 139

half() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding method), 162

half() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF method), 147

half() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding (method), 170
 half() (flood_forecast.transformer_xl.transformer_xl.TransformerXLMethod), 39
 half() (flood_forecast.transformer_xl.transformer_xl.TransformerXLMethod), 178
 haversine() (in module (flood_forecast.da_rnn.model.DARNN (method), 195
 23 load_state_dict() (flood_forecast.da_rnn.modules.Decoder (method), 213
 | load_state_dict() (flood_forecast.da_rnn.modules.Encoder (method), 205
 index() (flood_forecast.da_rnn.custom_types.DaRnnNet load_state_dict() (method), 192
 index() (flood_forecast.da_rnn.custom_types.TrainConfig load_state_dict() (method), 191
 index() (flood_forecast.da_rnn.custom_types.TrainData (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel (method), 71
 infer_on_torch_model() (in module load_state_dict() (flood_forecast.transformer_xl.lower_upper_config.AR (method), 87
 init_hidden() (in module load_state_dict() (flood_forecast.da_rnn.modules), 203
 init_memory() (flood_forecast.transformer_xl.transformer_xl.TrainConfig (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding (method), 176
 94
 initial_layer() (in module load_state_dict() (flood_forecast.transformer_xl.lower_upper_config), (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF (method), 79
 77
 interpolate_missing_values() (in module load_state_dict() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadS (method), 103
 19
 inverse_scale() (flood_forecast.preprocessing.pytorch_loaders.CSVDataLoader (flood_forecast.transformer_xl.transformer_basic.CustomTransformer (method), 34
 121
 inverse_scale() (flood_forecast.preprocessing.pytorch_loaders.CSVDataLoader (flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF (method), 129
 129
 J join_data() (in module load_state_dict() (flood_forecast.preprocessing.buil_dataset), (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer (method), 113
 21 load_state_dict() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock (method), 154
 L load_model() (flood_forecast.time_model.PyTorchForecast load_state_dict() (method), 15
 load_model() (flood_forecast.time_model.TimeSeriesModel (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention (method), 139
 15
 load_state_dict() load_state_dict() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding (method), 162
 (flood_forecast.custom.custom_opt.BertAdam (method), 68
 load_state_dict() load_state_dict() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF (method), 147
 (flood_forecast.custom.custom_opt.GaussianLoss (method), 54
 load_state_dict() load_state_dict() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding (method), 170
 (flood_forecast.custom.custom_opt.MAPELoss (method), 47
 load_state_dict() load_state_dict() (flood_forecast.transformer_xl.transformer_xl.TransformerXL (method), 178
 (flood_forecast.custom.custom_opt.QuantileLoss (method), 62

`loop_through()` (in module `flood_forecast.long_train`), 7
`loss_func` (`flood_forecast.da_rnn.custom_types.TrainConfig` attribute), 191

M

`main()` (in module `flood_forecast.long_train`), 7
`main()` (in module `flood_forecast.trainer`), 17
`make_column_names()` (in module `flood_forecast.preprocessing.data_converter`), 25
`make_config_file()` (in module `flood_forecast.long_train`), 7
`make_data()` (in module `flood_forecast.preprocessing.preprocess_da_rnn`), 27
`make_data_load()` (`flood_forecast.time_model.PyTorchForecast` method), 15
`make_data_load()` (`flood_forecast.time_model.TimeSeriesModel` method), 15
`make_gage_data_csv()` (in module `flood_forecast.preprocessing.preprocess_metadata`), 29
`make_temporal_features()` (in module `flood_forecast.preprocessing.temporal_feats`), 35
`make_usgs_data()` (in module `flood_forecast.preprocessing.process_usgs`), 31
`MAPELoss` (class in `flood_forecast.custom.custom_opt`), 45
`MAPELoss.to()` (in module `flood_forecast.custom.custom_opt`), 51
`MetaEmbedding` (class in `flood_forecast.transformer_xl.lower_upper_config`), 92
`MetaEmbedding.to()` (in module `flood_forecast.transformer_xl.lower_upper_config`), 99
`metric_dict()` (in module `flood_forecast.evaluator`), 3
`modules()` (`flood_forecast.custom.custom_opt.GaussianLoss` method), 55
`modules()` (`flood_forecast.custom.custom_opt.MAPELoss` method), 47
`modules()` (`flood_forecast.custom.custom_opt.QuantileLoss` method), 62
`modules()` (`flood_forecast.custom.custom_opt.RMSELoss` method), 39
`modules()` (`flood_forecast.da_rnn.model.DARNN` method), 195
`modules()` (`flood_forecast.da_rnn.modules.Decoder` method), 213
`modules()` (`flood_forecast.da_rnn.modules.Encoder` method), 205
`modules()` (`flood_forecast.transformer_xl.dummy_torch.DummyTorchModel` method), 71
`modules()` (`flood_forecast.transformer_xl.lower_upper_config.AR` method), 87
`modules()` (`flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding` method), 95
`modules()` (`flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward` method), 79
`modules()` (`flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple` method), 104
`modules()` (`flood_forecast.transformer_xl.transformer_basic.CustomTransformerBlock` method), 122
`modules()` (`flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFeedForward` method), 129
`modules()` (`flood_forecast.transformer_xl.transformer_basic.SimpleTransformerBlock` method), 114
`modules()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock` method), 155
`modules()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention` method), 140
`modules()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` method), 162
`modules()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward` method), 147
`modules()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedder` method), 170
`modules()` (`flood_forecast.transformer_xl.transformer_xl.TransformerXL` method), 178
`MultiAttnHeadSimple` (class in `flood_forecast.transformer_xl.multi_head_base`), 101
`MultiAttnHeadSimple.to()` (in module `flood_forecast.transformer_xl.multi_head_base`), 107, 108
`MultiHeadAttention` (class in `flood_forecast.transformer_xl.transformer_xl`), 137
`MultiHeadAttention.to()` (in module `flood_forecast.transformer_xl.transformer_xl`), 143, 144

N

`named_buffers()` (`flood_forecast.custom.custom_opt.GaussianLoss` method), 55
`named_buffers()` (`flood_forecast.custom.custom_opt.MAPELoss` method), 47
`named_buffers()` (`flood_forecast.custom.custom_opt.QuantileLoss` method), 62
`named_buffers()` (`flood_forecast.custom.custom_opt.RMSELoss` method), 40
`named_buffers()` (`flood_forecast.da_rnn.model.DARNN` method), 196
`named_buffers()` (`flood_forecast.da_rnn.modules.Decoder` method), 213

named_buffers() (flood_forecast.da_rnn.modules.Encoder method), 206
 named_buffers() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 72
 named_buffers() (flood_forecast.transformer_xl.lower_upper_config.AR method), 87
 named_buffers() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 95
 named_buffers() (flood_forecast.transformer_xl.lower_upper_config.PositionalEmbedding method), 80
 named_buffers() (flood_forecast.transformer_xl.multi_head_base.MultiHeadAttention method), 104
 named_buffers() (flood_forecast.transformer_xl.transformer_basic.CustomTransformer method), 122
 named_buffers() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEmbedding method), 130
 named_buffers() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer method), 114
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock method), 155
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention method), 140
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding method), 163
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding method), 147
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding method), 170
 named_buffers() (flood_forecast.transformer_xl.transformer_xl.TransformerXL method), 178
 named_children() (flood_forecast.custom.custom_opt.GaussianLoss method), 55
 named_children() (flood_forecast.custom.custom_opt.MAPELoss method), 48
 named_children() (flood_forecast.custom.custom_opt.QuantileLoss method), 63
 named_children() (flood_forecast.custom.custom_opt.RMSELoss method), 40
 named_children() (flood_forecast.da_rnn.model.DARNN method), 196
 named_children() (flood_forecast.da_rnn.modules.Decoder method), 214
 named_children() (flood_forecast.da_rnn.modules.Encoder method), 206
 named_children() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel method), 72
 named_children() (flood_forecast.transformer_xl.lower_upper_config.AR method), 88
 named_children() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding method), 95
 named_children() (flood_forecast.transformer_xl.lower_upper_config.PositionalEmbedding method), 80
 named_children() (flood_forecast.transformer_xl.multi_head_base.MultiHeadAttention method), 104
 named_children() (flood_forecast.transformer_xl.transformer_basic.CustomTransformer method), 122
 named_children() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEmbedding method), 130
 named_children() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer method), 114
 named_children() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock method), 155
 named_children() (flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention method), 140
 named_children() (flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding method), 163

named_modules() (flood_forecast.transformer_xl.transformer_xl.PositionwiseFF)
 method), 148
 named_modules() (flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding
 method), 171
 named_modules() (flood_forecast.transformer_xl.transformer_xl.TransformerXLtransformer_xl.transformer_xl.PositionalEmbedding
 method), 179
 named_parameters() (flood_forecast.custom.custom_opt.GaussianLoss
 method), 56
 named_parameters() (flood_forecast.custom.custom_opt.MAPELoss
 method), 48
 named_parameters() (flood_forecast.custom.custom_opt.QuantileLoss
 method), 63
 named_parameters() (flood_forecast.custom.custom_opt.RMSELoss
 method), 41
 named_parameters() (flood_forecast.da_rnn.model.DARNN
 method), 196
 named_parameters() (flood_forecast.da_rnn.modules.Decoder
 method), 214
 named_parameters() (flood_forecast.da_rnn.modules.Encoder
 method), 206
 named_parameters() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel
 method), 72
 named_parameters() (flood_forecast.transformer_xl.lower_upper_config.AR
 method), 88
 named_parameters() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbeddings
 method), 96
 named_parameters() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward
 method), 81
 named_parameters() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple
 method), 105
 named_parameters() (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder
 method), 123
 named_parameters() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding
 method), 130
 named_parameters() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer
 method), 115
 named_parameters() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock
 method), 156

P

parameters() (flood_forecast.custom.custom_opt.GaussianLoss
 method), 56
 parameters() (flood_forecast.custom.custom_opt.MAPELoss
 method), 48
 parameters() (flood_forecast.custom.custom_opt.QuantileLoss
 method), 64
 parameters() (flood_forecast.custom.custom_opt.RMSELoss
 method), 41
 parameters() (flood_forecast.da_rnn.model.DARNN
 method), 197
 parameters() (flood_forecast.da_rnn.modules.Decoder
 method), 214
 parameters() (flood_forecast.da_rnn.modules.Encoder
 method), 207
 parameters() (flood_forecast.transformer_xl.dummy_torch.DummyTorchModel
 method), 73
 parameters() (flood_forecast.transformer_xl.lower_upper_config.AR
 method), 88
 parameters() (flood_forecast.transformer_xl.lower_upper_config.MetaEmbeddings
 method), 96
 parameters() (flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward
 method), 81
 parameters() (flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple
 method), 105
 parameters() (flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder
 method), 123
 parameters() (flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding
 method), 130
 parameters() (flood_forecast.transformer_xl.transformer_basic.SimpleTransformer
 method), 115
 parameters() (flood_forecast.transformer_xl.transformer_xl.DecoderBlock
 method), 156

`parameters()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` `hook()` `method`), 164
`parameters()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFF` `hook()` `method`), 149
`parameters()` (`flood_forecast.transformer_xl.transformer_xl.StandardScaler` `hook()` `method`), 172
`parameters()` (`flood_forecast.transformer_xl.transformer_xl.register_backward_hook()` `hook()` `method`), 180
`plot_r2()` (in module `flood_forecast.evaluator`), 3
`PositionalEmbedding` (class in `flood_forecast.transformer_xl.transformer_xl`), 160
`PositionalEmbedding.to()` (in module `flood_forecast.transformer_xl.transformer_xl`), 166
`PositionwiseFeedForward` (class in `flood_forecast.transformer_xl.lower_upper_config`), 77
`PositionwiseFeedForward.to()` (in module `flood_forecast.transformer_xl.lower_upper_config`), 83
`PositionwiseFF` (class in `flood_forecast.transformer_xl.transformer_xl`), 145
`PositionwiseFF.to()` (in module `flood_forecast.transformer_xl.transformer_xl`), 151
`predict()` (in module `flood_forecast.da_rnn.train_da`), 187
`prep_train_data()` (in module `flood_forecast.da_rnn.train_da`), 187
`process_asos_csv()` (in module `flood_forecast.preprocessing.closest_station`), 23
`process_asos_data()` (in module `flood_forecast.preprocessing.closest_station`), 23
`process_intermediate_csv()` (in module `flood_forecast.preprocessing.process_usgs`), 31
`process_response_text()` (in module `flood_forecast.preprocessing.process_usgs`), 31
`PyTorchForecast` (class in `flood_forecast.time_model`), 15
Q
`QuantileLoss` (class in `flood_forecast.custom.custom_opt`), 60
`QuantileLoss.to()` (in module `flood_forecast.custom.custom_opt`), 66
R
`register_backward_hook()` (`flood_forecast.custom.custom_opt.GaussianLoss` `method`), 56
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` `hook()` `method`), 164
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFF` `hook()` `method`), 149
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.StandardScaler` `hook()` `method`), 64
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.register_backward_hook()` `hook()` `method`), 180
`register_backward_hook()` (`flood_forecast.custom.custom_opt.RMSELoss` `method`), 41
`register_backward_hook()` (`flood_forecast.da_rnn.model.DARNN` `method`), 197
`register_backward_hook()` (`flood_forecast.da_rnn.modules.Decoder` `method`), 215
`register_backward_hook()` (`flood_forecast.da_rnn.modules.Encoder` `method`), 207
`register_backward_hook()` (`flood_forecast.transformer_xl.dummy_torch.DummyTorchModel` `method`), 73
`register_backward_hook()` (`flood_forecast.transformer_xl.lower_upper_config.AR` `method`), 89
`register_backward_hook()` (`flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding` `method`), 96
`register_backward_hook()` (`flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF` `method`), 81
`register_backward_hook()` (`flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadS` `method`), 105
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_basic.CustomTransfo` `method`), 123
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_basic.SimplePosition` `method`), 131
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_basic.SimpleTransfo` `method`), 115
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock` `method`), 156
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention` `method`), 141
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` `hook()` `method`), 164
`register_backward_hook()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFF` `method`), 149

<code>register_backward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding</code> <code>method</code>), 172	<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention</code> <code>method</code>), 142
<code>register_backward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.TransformerXLFlood_forecast.transformer_xl.transformer_xl.PositionalEmbedding</code> <code>method</code>), 180	<code>register_buffer()</code> <code>method</code>), 165
<code>register_buffer()</code> (<code>flood_forecast.custom.custom_opt.GaussianLoss</code> <code>method</code>), 57	<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_xl.PositionwiseFF</code> <code>method</code>), 149
<code>register_buffer()</code> (<code>flood_forecast.custom.custom_opt.MAPELoss</code> <code>method</code>), 49	<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_xl.StandardWordEmb</code> <code>method</code>), 172
<code>register_buffer()</code> (<code>flood_forecast.custom.custom_opt.QuantileLoss</code> <code>method</code>), 64	<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_xl.TransformerXL</code> <code>method</code>), 180
<code>register_buffer()</code> (<code>flood_forecast.custom.custom_opt.RMSELoss</code> <code>method</code>), 42	<code>register_forward_hook()</code> (<code>flood_forecast.custom.custom_opt.GaussianLoss</code> <code>method</code>), 57
<code>register_buffer()</code> (<code>flood_forecast.da_rnn.model.DARNN</code> <code>method</code>), 198	<code>register_forward_hook()</code> (<code>flood_forecast.custom.custom_opt.MAPELoss</code> <code>method</code>), 50
<code>register_buffer()</code> (<code>flood_forecast.da_rnn.modules.Decoder</code> <code>method</code>), 215	<code>register_forward_hook()</code> (<code>flood_forecast.custom.custom_opt.QuantileLoss</code> <code>method</code>), 65
<code>register_buffer()</code> (<code>flood_forecast.da_rnn.modules.Encoder</code> <code>method</code>), 208	<code>register_forward_hook()</code> (<code>flood_forecast.custom.custom_opt.RMSELoss</code> <code>method</code>), 42
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.dummy_torch.DummyTorchModel</code> <code>method</code>), 73	<code>register_forward_hook()</code> (<code>flood_forecast.da_rnn.model.DARNN</code> <code>method</code>), 198
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.AR</code> <code>method</code>), 89	<code>register_forward_hook()</code> (<code>flood_forecast.da_rnn.modules.Decoder</code> <code>method</code>), 216
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding</code> <code>method</code>), 97	<code>register_forward_hook()</code> (<code>flood_forecast.da_rnn.modules.Encoder</code> <code>method</code>), 208
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF</code> <code>method</code>), 82	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.dummy_torch.DummyTorchModel</code> <code>method</code>), 74
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple</code> <code>method</code>), 106	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.AR</code> <code>method</code>), 90
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_basic.CustomTransformer</code> <code>method</code>), 124	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding</code> <code>method</code>), 97
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF</code> <code>method</code>), 132	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF</code> <code>method</code>), 82
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimpleTransformer</code> <code>method</code>), 116	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadS</code> <code>method</code>), 106
<code>register_buffer()</code> (<code>flood_forecast.transformer_xl.transformer_xl.DecoderBlock</code> <code>method</code>), 157	<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.CustomTransfo</code> <code>method</code>), 124

<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimplePositionwiseFF</code> <code>method</code>), 132	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF</code> <code>method</code>), 82
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimpleTransformer</code> <code>method</code>), 116	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadS</code> <code>method</code>), 106
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.DecoderBlock</code> <code>method</code>), 157	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.CustomTransfo</code> <code>method</code>), 124
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.MultiHeadAttn</code> <code>method</code>), 142	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimplePosition</code> <code>method</code>), 132
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding</code> <code>method</code>), 165	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_basic.SimpleTransfo</code> <code>method</code>), 117
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.PositionwiseFF</code> <code>method</code>), 150	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.DecoderBlock</code> <code>method</code>), 158
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding</code> <code>method</code>), 173	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention</code> <code>method</code>), 142
<code>register_forward_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.TransformerXL</code> <code>method</code>), 181	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.PositionalEmbedd</code> <code>method</code>), 165
<code>register_forward_pre_hook()</code> (<code>flood_forecast.custom.custom_opt.GaussianLoss</code> <code>method</code>), 57	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.PositionwiseFF</code> <code>method</code>), 150
<code>register_forward_pre_hook()</code> (<code>flood_forecast.custom.custom_opt.MAPELoss</code> <code>method</code>), 50	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.StandardWordEmb</code> <code>method</code>), 173
<code>register_forward_pre_hook()</code> (<code>flood_forecast.custom.custom_opt.QuantileLoss</code> <code>method</code>), 65	<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.transformer_xl.TransformerXL</code> <code>method</code>), 181
<code>register_forward_pre_hook()</code> (<code>flood_forecast.custom.custom_opt.RMSELoss</code> <code>method</code>), 42	<code>register_parameter()</code> (<code>flood_forecast.custom.custom_opt.GaussianLoss</code> <code>method</code>), 58
<code>register_forward_pre_hook()</code> (<code>flood_forecast.da_rnn.model.DARNN</code> <code>method</code>), 198	<code>register_parameter()</code> (<code>flood_forecast.custom.custom_opt.MAPELoss</code> <code>method</code>), 50
<code>register_forward_pre_hook()</code> (<code>flood_forecast.da_rnn.modules.Decoder</code> <code>method</code>), 216	<code>register_parameter()</code> (<code>flood_forecast.custom.custom_opt.QuantileLoss</code> <code>method</code>), 65
<code>register_forward_pre_hook()</code> (<code>flood_forecast.da_rnn.modules.Encoder</code> <code>method</code>), 208	<code>register_parameter()</code> (<code>flood_forecast.custom.custom_opt.RMSELoss</code> <code>method</code>), 43
<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.dummy_torch.DummyTorchModel</code> <code>method</code>), 74	<code>register_parameter()</code> (<code>flood_forecast.da_rnn.model.DARNN</code> <code>method</code>), 198
<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.AR</code> <code>method</code>), 90	<code>register_parameter()</code> (<code>flood_forecast.da_rnn.modules.Decoder</code> <code>method</code>), 216
<code>register_forward_pre_hook()</code> (<code>flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding</code> <code>method</code>), 97	<code>register_parameter()</code> (<code>flood_forecast.da_rnn.modules.Encoder</code> <code>method</code>), 208

`register_parameter()` (`flood_forecast.da_rnn.modules.Encoder`
`(flood_forecast.transformer_xl.dummy_torch.DummyTorchModule)`, 209
`method`), 74 `requires_grad_()` (`flood_forecast.transformer_xl.dummy_torch.DummyTorchModule`), 75
`register_parameter()` (`flood_forecast.transformer_xl.lower_upper_config.AR`
`method`), 90 `requires_grad_()` (`flood_forecast.transformer_xl.lower_upper_config.AR`
`method`), 90
`register_parameter()` (`flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding`
`method`), 98 `requires_grad_()` (`flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding`
`method`), 83
`register_parameter()` (`flood_forecast.transformer_xl.lower_upper_config.PositionalEncodingFeedForward`
`method`), 83 `requires_grad_()` (`flood_forecast.transformer_xl.lower_upper_config.PositionalEncodingFeedForward`
`method`), 107
`register_parameter()` (`flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple`
`method`), 107 `requires_grad_()` (`flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple`
`method`), 125
`register_parameter()` (`flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder`
`method`), 125 `requires_grad_()` (`flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder`
`method`), 117
`register_parameter()` (`flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding`
`method`), 132 `requires_grad_()` (`flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding`
`method`), 132
`register_parameter()` (`flood_forecast.transformer_xl.transformer_basic.SimpleTransformer`
`method`), 117 `requires_grad_()` (`flood_forecast.transformer_xl.transformer_basic.SimpleTransformer`
`method`), 143
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock`
`method`), 158 `requires_grad_()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock`
`method`), 150
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention`
`method`), 143 `requires_grad_()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention`
`method`), 181
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding`
`method`), 166 `reset_length()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding`
`method`), 176
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding`
`method`), 173 `RMSELoss` (`class in flood_forecast.custom.custom_opt`), 37
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding`
`method`), 173 `RMSELoss.to()` (`in module flood_forecast.custom.custom_opt`), 43
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding`
`method`), 173 `save_model()` (`flood_forecast.time_model.PyTorchForecast`
`method`), 15
`register_parameter()` (`flood_forecast.transformer_xl.transformer_xl.TransformerXL`
`method`), 181 `save_model()` (`flood_forecast.time_model.TimeSeriesModel`
`method`), 15
`requires_grad_()` (`flood_forecast.custom.custom_opt.GaussianLoss`
`method`), 58 `save_model_checkpoint()` (`flood_forecast.utils.EarlyStopper` `method`),
`requires_grad_()` (`flood_forecast.custom.custom_opt.MAPELoss`
`method`), 50 `save_or_show_plot()` (`in module flood_forecast.da_rnn.utils`), 189
`requires_grad_()` (`flood_forecast.custom.custom_opt.QuantileLoss`
`method`), 65 `setup_log()` (`in module flood_forecast.da_rnn.utils`), 89
`requires_grad_()` (`flood_forecast.custom.custom_opt.RMSELoss`
`method`), 43 `share_memory()` (`flood_forecast.custom.custom_opt.GaussianLoss`
`method`), 58
`requires_grad_()` (`flood_forecast.da_rnn.model.DARNN`
`method`), 199 `share_memory()` (`flood_forecast.custom.custom_opt.MAPELoss`
`method`), 50
`requires_grad_()` (`flood_forecast.da_rnn.modules.Decoder`
`method`), 216

`share_memory()` (`flood_forecast.custom.custom_opt.QuantileLoss`), 66
`share_memory()` (`flood_forecast.custom.custom_opt.RMSELoss`), 43
`share_memory()` (`flood_forecast.da_rnn.model.DARNNStandardWordEmbedding.to()`), 199
`share_memory()` (`flood_forecast.da_rnn.modules.Decoder`), 216
`share_memory()` (`flood_forecast.da_rnn.modules.Encoder`), 209
`share_memory()` (`flood_forecast.transformer_xl.dummy_torch.DummyTorchModel`), 75
`share_memory()` (`flood_forecast.transformer_xl.lower_upper_config.AR`), 90
`share_memory()` (`flood_forecast.transformer_xl.lower_upper_config.Meta`), 98
`share_memory()` (`flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward`), 83
`share_memory()` (`flood_forecast.transformer_xl.multi_head_base.MultiHeadAttention`), 107
`share_memory()` (`flood_forecast.transformer_xl.transformer_basic.CustomPositionalEncoding`), 125
`share_memory()` (`flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding`), 133
`share_memory()` (`flood_forecast.transformer_xl.transformer_basic.StandardWordEmbedding`), 117
`share_memory()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock`), 158
`share_memory()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention`), 143
`share_memory()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward`), 166
`share_memory()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding`), 174
`share_memory()` (`flood_forecast.transformer_xl.transformer_xl.TransformerXL`), 182
`SimplePositionalEncoding` (class in `flood_forecast.transformer_xl.transformer_basic`), 127
`SimplePositionalEncoding.to()` (in module `flood_forecast.transformer_xl.transformer_basic`), 133
`SimpleTransformer` (class in `flood_forecast.transformer_xl.transformer_basic`), 111
`SimpleTransformer.to()` (in module `flood_forecast.transformer_xl.transformer_basic`), 118
`split_on_letter()` (in module `flood_forecast.long_train`), 7
`split_on_na_chunks()` (in module `flood_forecast.preprocessing.interpolate_preprocess`), 68
`stream_baseline()` (in module `flood_forecast.preprocessing.interpolate_preprocess`), 68

flood_forecast.evaluator), 3

T

T (*flood_forecast.da_rnn.custom_types.TrainConfig* attribute), 191

T_destination (*flood_forecast.custom.custom_opt.GaussianLoss* attribute), 52

T_destination (*flood_forecast.custom.custom_opt.MAPELoss* attribute), 45

T_destination (*flood_forecast.custom.custom_opt.QuantileLoss* attribute), 60

T_destination (*flood_forecast.custom.custom_opt.RMSELoss* attribute), 37

T_destination (*flood_forecast.da_rnn.model.DARNN* attribute), 193

T_destination (*flood_forecast.da_rnn.modules.Decoder* attribute), 211

T_destination (*flood_forecast.da_rnn.modules.Encoder* attribute), 203

T_destination (*flood_forecast.transformer_xl.dummy_torch.DummyTorchModel* attribute), 69

T_destination (*flood_forecast.transformer_xl.lower_upper_config.AR* attribute), 85

T_destination (*flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding* attribute), 92

T_destination (*flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward* attribute), 77

T_destination (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple* attribute), 101

T_destination (*flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder* attribute), 119

T_destination (*flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding* attribute), 127

T_destination (*flood_forecast.transformer_xl.transformer_basic.StandardTransformer* attribute), 111

T_destination (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock* attribute), 153

T_destination (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention* attribute), 137

T_destination (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding* attribute), 160

T_destination (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward* attribute), 145

T_destination (*flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding* attribute), 168

T_destination (*flood_forecast.transformer_xl.transformer_xl.TransformerXL* attribute), 176

targs (*flood_forecast.da_rnn.custom_types.TrainData* attribute), 191

TimeSeriesModel (class in *flood_forecast.time_model*), 15

to() (*flood_forecast.custom.custom_opt.GaussianLoss* method), 58

to() (*flood_forecast.custom.custom_opt.MAPELoss* method), 51

to() (*flood_forecast.custom.custom_opt.QuantileLoss* method), 66

to() (*flood_forecast.custom.custom_opt.RMSELoss* method), 43

to() (*flood_forecast.da_rnn.model.DARNN* method), 199

to() (*flood_forecast.da_rnn.modules.Decoder* method), 217

to() (*flood_forecast.da_rnn.modules.Encoder* method), 209

to() (*flood_forecast.transformer_xl.dummy_torch.DummyTorchModel* method), 75

to() (*flood_forecast.transformer_xl.lower_upper_config.AR* method), 91

to() (*flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding* method), 98

to() (*flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward* method), 83

to() (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple* method), 107

to() (*flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder* method), 125

to() (*flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding* method), 133

to() (*flood_forecast.transformer_xl.transformer_basic.SimpleTransformer* method), 118

to() (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock* method), 159

to() (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention* method), 143

to() (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding* method), 166

to() (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFeedForward* method), 151

to() (*flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding* method), 174

to() (*flood_forecast.transformer_xl.transformer_xl.TransformerXL* method), 182

torch_single_train() (in module *flood_forecast.pytorch_training*), 13

train() (*flood_forecast.custom.custom_opt.GaussianLoss* method), 50

train() (*flood_forecast.custom.custom_opt.MAPELoss* method), 52

train() (*flood_forecast.custom.custom_opt.QuantileLoss* method), 67

train() (*flood_forecast.custom.custom_opt.RMSELoss* method), 44

train() (*flood_forecast.da_rnn.model.DARNN* method), 200

train() (*flood_forecast.da_rnn.modules.Decoder* method), 218

`train()` (`flood_forecast.da_rnn.modules.Encoder` `type()` (`flood_forecast.custom.custom_opt.QuantileLoss` `method`), 210 `method`), 67
`train()` (`flood_forecast.transformer_xl.dummy_torch.DummyTorchModel` `method`), 76 `method`), 44
`train()` (`flood_forecast.transformer_xl.lower_upper_config.AR` `method`), 92 `method`), 200
`train()` (`flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding` `method`), 100 `method`), 218
`train()` (`flood_forecast.transformer_xl.lower_upper_config.PositionwiseFF` `method`), 84 `method`), 210
`train()` (`flood_forecast.transformer_xl.multi_head_base.MultiAttnHead` `method`), 108 `method`), 76
`train()` (`flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder` `method`), 127 `method`), 92
`train()` (`flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding` `method`), 134 `method`), 100
`train()` (`flood_forecast.transformer_xl.transformer_basic.SimpleTransformer` `method`), 119 `method`), 84
`train()` (`flood_forecast.transformer_xl.transformer_xl.DecoderBlock` `method`), 160 `method`), 109
`train()` (`flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention` `method`), 144 `method`), 127
`train()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` `method`), 167 `method`), 134
`train()` (`flood_forecast.transformer_xl.transformer_xl.PositionwiseFF` `method`), 152 `method`), 119
`train()` (`flood_forecast.transformer_xl.transformer_xl.StandardWordEmbedding` `method`), 175 `method`), 160
`train()` (`flood_forecast.transformer_xl.transformer_xl.TransformerXL` `method`), 183 `method`), 145
`train()` (`in module flood_forecast.da_rnn.train_da`), `type()` (`flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding` `method`), 187 `method`), 167
`train_function()` (`in module flood_forecast.trainer`), 17 `method`), 152
`train_iteration()` (`in module flood_forecast.da_rnn.train_da`), 187 `method`), 175
`train_size()` (`flood_forecast.da_rnn.custom_types.TrainConfig` `attribute`), 191 `method`), 183
`train_transformer_style()` (`in module flood_forecast.pytorch_training`), 13
TrainConfig (`class in flood_forecast.da_rnn.custom_types`), 191
TrainData (`class in flood_forecast.da_rnn.custom_types`), 191
TransformerXL (`class in flood_forecast.transformer_xl.transformer_xl`), 175
TransformerXL.to() (`in module flood_forecast.transformer_xl.transformer_xl`), 182
`type()` (`flood_forecast.custom.custom_opt.GaussianLoss` `method`), 60
`type()` (`flood_forecast.custom.custom_opt.MAPELoss` `method`), 52
U
`update_memory()` (`flood_forecast.transformer_xl.transformer_xl.TransformerXL` `method`), 176
`upload_file()` (`in module flood_forecast.gcp_integration.basic_utils`), 185
`upload_gcs()` (`flood_forecast.time_model.PyTorchForecast` `method`), 15
`upload_gcs()` (`flood_forecast.time_model.TimeSeriesModel` `method`), 15
V
`variable_forecast_layer()` (`in module flood_forecast.transformer_xl.lower_upper_config`), 77

W

`wandb_init()` (*flood_forecast.time_model.PyTorchForecast* *method*), 175
method), 15
`wandb_init()` (*flood_forecast.time_model.TimeSeriesModel* *method*), 183
method), 15
`warmup_constant()` (*in module*
flood_forecast.custom.custom_opt), 37
`warmup_cosine()` (*in module*
flood_forecast.custom.custom_opt), 37
`warmup_linear()` (*in module*
flood_forecast.custom.custom_opt), 37

Z

`zero_grad()` (*flood_forecast.custom.custom_opt.BertAdam*
method), 68
`zero_grad()` (*flood_forecast.custom.custom_opt.GaussianLoss*
method), 60
`zero_grad()` (*flood_forecast.custom.custom_opt.MAPELoss*
method), 52
`zero_grad()` (*flood_forecast.custom.custom_opt.QuantileLoss*
method), 67
`zero_grad()` (*flood_forecast.custom.custom_opt.RMSELoss*
method), 45
`zero_grad()` (*flood_forecast.da_rnn.model.DARNN*
method), 201
`zero_grad()` (*flood_forecast.da_rnn.modules.Decoder*
method), 218
`zero_grad()` (*flood_forecast.da_rnn.modules.Encoder*
method), 211
`zero_grad()` (*flood_forecast.transformer_xl.dummy_torch.DummyTorchModel*
method), 76
`zero_grad()` (*flood_forecast.transformer_xl.lower_upper_config.AR*
method), 92
`zero_grad()` (*flood_forecast.transformer_xl.lower_upper_config.MetaEmbedding*
method), 100
`zero_grad()` (*flood_forecast.transformer_xl.lower_upper_config.PositionwiseFeedForward*
method), 85
`zero_grad()` (*flood_forecast.transformer_xl.multi_head_base.MultiAttnHeadSimple*
method), 109
`zero_grad()` (*flood_forecast.transformer_xl.transformer_basic.CustomTransformerDecoder*
method), 127
`zero_grad()` (*flood_forecast.transformer_xl.transformer_basic.SimplePositionalEncoding*
method), 135
`zero_grad()` (*flood_forecast.transformer_xl.transformer_basic.SimpleTransformer*
method), 119
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.DecoderBlock*
method), 160
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.MultiHeadAttention*
method), 145
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.PositionalEmbedding*
method), 168
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.PositionwiseFF*
method), 152
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.StandardWo*
method), 175
`zero_grad()` (*flood_forecast.transformer_xl.transformer_xl.Transforme*
method), 183