Flow Forecast

Release 0.0.1

Aug 11, 2020

General:

CHAPTER 1

Utilities

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): It 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.TimeSeriesMod df: pandas.core.frame.DataFrame, test_data: flood_forecast.preprocessing.pytorch_loaders.CS history_dim: torch.Tensor, forecast_length: int, hours_to_forecast: int*) → torch.Tensor

flood_forecast.evaluator.**generate_decoded_predictions**(*model: Type[flood_forecast.time_model.TimeSeriesModel], test_data: flood_forecast.preprocessing.pytorch_loaders.CSVTest 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, decoder_params: Dict[KT, VT], num_prediction_samples: int*) → numpy.ndarray

CHAPTER 3

Long Train

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

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

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

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

CHAPTER 6

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, forward_params: Dict[KT, VT] = {}*) → float

flood_forecast.pytorch_training.**compute_validation**(*validation_loader: torch.utils.data.dataloader.DataLoader, model, epoch: int, sequence_size: int, criterion: Type[torch.nn.modules.loss._Loss], device: torch.device, decoder_structure=False, 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
Intializes a data loader based on the provided data_path. This may be as simple as a pandas dataframe or as complex as a custom PyTorch data loader.

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

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

wandb_init()

class flood_forecast.time_model.**PyTorchForecast**(*model_base: str, training_data, validation_data, test_data, params_dict: Dict[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_load(*data_path: str, dataset_params: Dict[KT, VT], loader_type: str, the_class='default'*)

Intializes a data loader based on the provided data_path. This may be as simple as a pandas dataframe or as complex as a custom PyTorch data loader.

CHAPTER 8

Trainer

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.

CHAPTER 9

Interpolate Preprocessing

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

Basic function to fix intial 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: pandas.core.frame.DataFrame*) → None

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

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

CHAPTER 10

Build Dataset

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

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

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

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

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

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

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

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*) → datetime.datetime

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

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

flood_forecast.preprocessing.closest_station.**process_asos_data**(*file_path: str*, *base_url: str*) → Dict[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: datetime.datetime*, *end_date: datetime.datetime*, *site_number: str*) → pandas.core.frame.DataFrame

flood_forecast.preprocessing.process_usgs.**process_response_text**(*file_name: str*) → Tuple[str, Dict[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: pandas.core.frame.DataFrame) -> (<class 'pandas.core.frame.DataFrame'>, <class 'int'>, <class 'int'>, <class 'int'>*)

CHAPTER 16

PyTorch Loaders

class flood_forecast.preprocessing.pytorch_loaders.**CSVDataLoader**(*file_path: str, forecast_history: int, forecast_length: int, target_col: List[T], relevant_cols: List[T], scaling=None, start_stamp: int = 0, end_stamp: int = None, interpolate_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 reccomended) If provided should be a subclass of sklearn.base.BaseEstimator

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

or testing supply these

Parameters **int** (*end_stamp*) – Optional if you want to only use part of a CSV for training, validation, or testing supply these

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*, *forecast_total: int*, *use_real_precip=True*, *use_real_temp=True*, *target_supplied=True*, *interpolate=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: pandas.core.frame.DataFrame*) → pandas.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*)

CHAPTER 18

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, `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)
```

(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() → 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, `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[[...], 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, `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[[...], 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-2-6fdbc26b2629

`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() → 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, `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[[...], 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.Tensor` s.

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, `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)
```

(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, `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.

CHAPTER 20

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, `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)
```

(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, `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[[...], 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*, *output_dim*, *predictor_number*, *predictor_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, `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[[...], 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]*) → 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 21

Simple Multi Attention Head Model

class flood_forecast.transformer_xl.multi_head_base.**MultiAttnHeadSimple**(*number_time_series: int*, *seq_len=10*, *output_seq_len=None*, *d_model=128*, *num_heads=8*, *forecast_length=None*, *dropout=0.1*, *sigmoid=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[[...], 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, `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[[...], 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*, *output_seq_length: int*, *n_time_series: int*, *d_model=128*, *output_dim=1*, *n_layers_encoder=*, *forward_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, `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[[...], 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]*) → 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.**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, `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[[...], 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*, *dropouta: 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, `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[[...], 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.**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() → 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, `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[[...], 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*, *dropouta=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, `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[[...], 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_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, `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)
```

(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() → 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, `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[[...], 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*, *dropouta=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() → 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, `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[[...], 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*, *upload_name: str*, *client: google.cloud.storage.client.Client*)

flood_forecast.gcp_integration.basic_utils.**create_file_environ**()

CHAPTER 25

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*, *decoder_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*)

CHAPTER 27

Custom Types

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.

CHAPTER 28

Model

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, `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[[...], 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 29

Modules

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, `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[[...], 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() → 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, `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[[...], 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

Python Module Index

f

Index

A

B

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W

Z