torchtable Documentation

Keita Kurita

Dec 23, 2018

Package Reference

Documentation is still a work in progress, so please bear with me while I get things ready! In the meantime, I've done my best to document the code so far, so feel free to take a look at the source code or raise an issue on GitHub.

CHAPTER 1

torchtable.utils module

1.1 Module contents

torchtable.utils.**apply_oneormore**(*func, x: Union[T, Iterable[T]]*) → Union[T, Iterable[T]]
Returns multiple outputs for multiple inputs and a single output for a single input, applying the same function in either case.

torchtable.utils.**expand**(*a: Union[T, Iterable[T]], n: int*) → Iterable
Turn 1 or n objects to n objects by repeating if necessary

torchtable.utils.**flat_filter**(*itr: Iterable[Union[T, Iterable[T]]], predicate: Callable[T, bool]*) → Iterable[T]

torchtable.utils.**fold_oneormore**(*func: Callable[[T, T], T], x: Union[T, Iterable[T]], init: T*) → Any
Aggregates multiple inputs by folding, simply applies for a single input.

torchtable.utils.**to_numpy_array**(*x: Union[pandas.core.series.Series, numpy.core.multiarray.array]*) → numpy.core.multiarray.array
Extracts numpy array from an ArrayLike object

torchtable.utils.**with_default**(*x: Optional[T], default: T*) → T

CHAPTER 2

torchtable.operator module

2.1 Module contents

class torchtable.operator.**Operator**

Bases: object

Base class for all operators. Operators can be chained together by piping their outputs to new operators or hooking operators to other operators. Any number of operators can be chained to become a pipeline, which is itself just another operator. Subclasses should implement the *apply* method that defines the operation performed by the operator.

Example

```
>>> class TimesThree(Operator):
...     def apply(self, x):
...         return x * 3
>>> op = TimeThree()
>>> op(4) # 4 * 3 = 12
... 12
```

```
>>> class Square(Operator):
...     def apply(self, x):
            return x ** 2
>>> op = TimesThree() > Square()
>>> op(2) # (2 * 3) ** 2 = 36
... 36
```

apply(*x: Any*, *train=True*) → Any

Takes output of previous stage in the pipeline and produces output. Override in subclasses.

Parameters

- **train** – If true, this operator will "train" on the input.

- **other words, the internal parameters of this operator may change to fit the given input.** (*In*) –

hook (*op: torchtable.operator.core.Operator*) → torchtable.operator.core.Operator
Connect an operator to the *beginning* of this pipeline. Returns self.

pipe (*op: torchtable.operator.core.Operator*) → torchtable.operator.core.Operator
Connect an operator after this operator. Returns the connected operator.

class torchtable.operator.**LambdaOperator** (*func: Callable[T, T]*)
Bases: torchtable.operator.core.Operator

Generic operator for stateless operation.

Parameters **func** – Function to apply to input.

apply (*x: T*, *train=True*) → Any
Takes output of previous stage in the pipeline and produces output. Override in subclasses.

Parameters

- **train** – If true, this operator will "train" on the input.
- **other words, the internal parameters of this operator may change to fit the given input.** (*In*) –

class torchtable.operator.**TransformerOperator** (*transformer*)
Bases: torchtable.operator.core.Operator

Wrapper for any stateful transformer with fit and transform methods.

Parameters **transformer** – Any object with a *fit* and *transform* method.

Example

```
>>> op = TransformerOperator(sklearn.preprocessing.StandardScaler())
```

apply (*x: Any*, *train=True*)
Takes output of previous stage in the pipeline and produces output. Override in subclasses.

Parameters

- **train** – If true, this operator will "train" on the input.
- **other words, the internal parameters of this operator may change to fit the given input.** (*In*) –

build (*x: Any*) → None

class torchtable.operator.**Normalize** (*method: Optional[str]*)
Bases: torchtable.operator.core.TransformerOperator

Normalizes a numeric field.

Parameters

- **method** – Method of normalization (choose from the following):
- **None** (–) – No normalization will be applied (same as noop)
- **'Gaussian'** (–) – Subtracts mean and divides by the standard deviation
- **'RankGaussian'** (–) – Assigns elements to a Gaussian distribution based on their rank.

class torchtable.operator.**FillMissing**(*method: Union[Callable, str]*)
Bases: torchtable.operator.core.TransformerOperator

Fills missing values according to *method*

Parameters

- **method** – Method of filling missing values. Options:
- **None** (–) – Do not fill missing values
- **'median'** (–) – Fill with median
- **'mean'** (–) – Fill with mean
- **'mode'** (–) – Fill with mode. Effective for categorical fields.
- **–** (*any callable*) – The output of the callable will be used to fill the missing values

class torchtable.operator.**Vocab**(*min_freq=0*, *max_features=None*, *handle_unk: Optional[bool] = False*, *nan_as_unk=False*)
Bases: object

Mapping from category to integer id

fit(*x: pandas.core.series.Series*) → torchtable.operator.core.Vocab
Construct the mapping

transform(*x: pandas.core.series.Series*) → pandas.core.series.Series

class torchtable.operator.**Categorize**(*min_freq: int = 0*, *max_features: Optional[int] = None*, *handle_unk: Optional[bool] = None*)
Bases: torchtable.operator.core.TransformerOperator

Converts categorical data into integer ids

Parameters

- **min_freq** – Minimum frequency required for a category to receive a unique id. Any categories with a lower frequency will be treated as unknown categories.
- **max_features** – Maximum number of unique categories to store. If larger than the number of actual categories, the categories with the highest frequencies will be chosen. If None, there will be no limit on the number of categories.
- **handle_unk** – Whether to allocate a unique id to unknown categories. If you expect to see categories that you did not encounter in your training data, you should set this to True. If None, handle_unk will be set to True if min_freq > 0 or max_features is not None, otherwise it will be False.

vocab_size

class torchtable.operator.**ToTensor**(*dtype: torch.dtype*)
Bases: torchtable.operator.core.Operator

Convert input to a *torch.tensor*

Parameters **dtype** – The dtype of the output tensor

apply(*x: Union[pandas.core.series.Series, numpy.core.multiarray.array], device: Optional[torch.device] = None, train=True*) → None._VariableFunctions.tensor
Takes output of previous stage in the pipeline and produces output. Override in subclasses.

Parameters

- **train** – If true, this operator will "train" on the input.

- **other words, the internal parameters of this operator may change to fit the given input.** (*In*) –

exception torchtable.operator.**UnknownCategoryError**
Bases: ValueError

CHAPTER 3

torchtable.field module

3.1 Module contents

class torchtable.field.**Field**(*pipeline: torchtable.operator.core.Operator*, *name: Optional[str] = None*, *is_target: bool = False*, *continuous: bool = True*, *categorical: bool = False*, *cardinality: Optional[int] = None*, *batch_pipeline: Optional[torchtable.operator.core.Operator] = None*, *dtype: Optional[torch.dtype] = None*, *metadata: dict = {}*)

Bases: `object`

A single field in the output mini batch. A Field acts as a continaer for all relevant information regarding an output in the output mini batch. Primarily, it stores a pipeline to apply to a column/set of columns in the input. It also stores a pipeline for converting the input batch to an appropriate type for the downstream model (generally a torch.tensor). This class can directly be instantiated with a custom pipeline but is generally used as a subclass for other fields.

Example

```
>>> fld = Field(LambdaOperator(lambda x: x + 1) > LambdaOperator(lambda x: x **␣
↪2))
>>> fld.transform(1)
... 9
```

Parameters

- **pipeline** – An operator representing the set of operations mapping the input column to the output. This transformation will be applied during the construction of the dataset. If the pipeline is resource intensive and applying it all at once is unrealistic, consider deferring some of the processing to *batch_pipeline*.
- **is_target** – Whether the field is an input or target field. Affects default batching behavior.

- **continuous** – Whether the output is continuous.
- **categorical** – Whether the output is categorical/discrete.
- **batch_pipeline** – The transformation to apply to this field during batching. By default, this will simply be an operation to transform the input to a tensor to feed to the model. This can be set to any Operator that the user wishes so that arbitrary transformations (e.g. padding, noising) can be applied during data loading.
- **dtype** – The output tensor dtype. Only relevant when batch_pipeline is None (using the default pipeline).
- **metadata** – Additional data about the field to store. Use cases include adding data about model parameters (e.g. size of embeddings for this field).

cardinality
Relevant for categorical data. For custom fields, the cardinality must be passed explicity.

index(*example: Union[pandas.core.series.Series, numpy.core.multiarray.array], idx*) → Union[pandas.core.series.Series, numpy.core.multiarray.array]
Wrapper for indexing. The field must provide the ability to index via a list for batching later on.

transform(*x: pandas.core.series.Series, train=True*) → Union[pandas.core.series.Series, numpy.core.multiarray.array]
Method to process the input column during construction of the dataset. Kwargs:

train: If true, this transformation may change some internal parameters of the pipeline.
For instance, if there is a normalization step in the pipeline, the mean and std will be computed on the current input. Otherwise, the pipeline will use statistics computed in the past.

transform_batch(*x: Union[pandas.core.series.Series, numpy.core.multiarray.array], device: Optional[torch.device] = None, train: bool = True*) → None._VariableFunctions.tensor
Method to process batch input during loading of the dataset.

class torchtable.field.**IdentityField**(*name=None, is_target=False, continuous=True, categorical=False, metadata={}*)
Bases: torchtable.field.core.Field

A field that does not modify the input.

class torchtable.field.**NumericField**(*name=None, fill_missing='median', normalization='Gaussian', is_target=False, metadata={}*)
Bases: torchtable.field.core.Field

A field corresponding to a continous, numerical output (e.g. price, distance, etc.)

Parameters

- **fill_missing** – The method of filling missing values. See the *FillMissing* operator for details.
- **normalization** – The method of normalization. See the *Normalize* operator for details.

class torchtable.field.**CategoricalField**(*name=None, min_freq=0, max_features=None, handle_unk=None, is_target=False, metadata: dict = {}*)
Bases: torchtable.field.core.Field

A field corresponding to a categorica, discrete output (e.g. id, group, gender)

Parameters **the Categorize operator for more details.** (*See*) –

cardinality
The number of unique outputs.

transform(*x: pandas.core.series.Series*, *train=True*) → Union[pandas.core.series.Series, numpy.core.multiarray.array]
Method to process the input column during construction of the dataset. Kwargs:

train: If true, this transformation may change some internal parameters of the pipeline.
For instance, if there is a normalization step in the pipeline, the mean and std will be computed on the current input. Otherwise, the pipeline will use statistics computed in the past.

class torchtable.field.**DatetimeFeatureField**(*func: Callable[pandas.core.series.Series, pandas.core.series.Series], fill_missing: Optional[str] = None, name=None, is_target=False, continuous=False, metadata: dict = {}*)
Bases: torchtable.field.core.Field

A generic field for constructing features from datetime columns. :param func: Feature construction function

class torchtable.field.**DayofWeekField**(***kwargs*)
Bases: torchtable.field.datetime.DatetimeFeatureField

class torchtable.field.**DayField**(***kwargs*)
Bases: torchtable.field.datetime.DatetimeFeatureField

class torchtable.field.**MonthStartField**(***kwargs*)
Bases: torchtable.field.datetime.DatetimeFeatureField

class torchtable.field.**MonthEndField**(***kwargs*)
Bases: torchtable.field.datetime.DatetimeFeatureField

class torchtable.field.**HourField**(***kwargs*)
Bases: torchtable.field.datetime.DatetimeFeatureField

torchtable.field.**date_fields**(***kwargs*) → List[torchtable.field.datetime.DatetimeFeatureField]
The default set of fields for feature engineering using a field with date information

torchtable.field.**datetime_fields**(***kwargs*) → List[torchtable.field.datetime.DatetimeFeatureField]
The default set of fields for feature engineering using a field with date and time information

class torchtable.field.**FieldCollection**(**args*, *flatten: bool = False*, *namespace: Optional[str] = None*)
Bases: list

A list of fields with some auxillary methods.

Parameters **flatten** – If set to True, each field in this collection will be mapped to one key in the batch/dataset. Otherwise, each field in this collection will be mapped to an entry in a list for the same key in the batch/dataset.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

name

set_namespace(*nm: str*) → None
Set names of inner fields as well

transform(**args*, ***kwargs*) → list
Applies transform with each field and returns a list

CHAPTER 4

torchtable.dataset module

4.1 Module contents

class torchtable.dataset.**TabularDataset**(*examples: Dict[Union[str, List[str]], Union[T, Iterable[T]]], fields: torchtable.dataset.core.FieldDict, train=True*)

Bases: torch.utils.data.dataset.Dataset

A dataset for tabular data.

Parameters

- **fields** – A dictionary mapping from a column/columns in the raw data to a Field/Fields. To specify multiple columns as input, use a tuple of column names. To map a single column to multiple fields, use a list of fields. Each field will be mapped to a single entry in the processed dataset.
- **train** – Whether this dataset is the training set. This affects whether the fields will fit the given data.

classmethod from_csv(*fname: str, fields: Dict[Union[str, List[str]], Union[torchtable.field.core.Field, torchtable.field.core.FieldCollection, Collection[torchtable.field.core.Field]]], train=True, csv_read_params: dict = {}*) → torchtable.dataset.core.TabularDataset

Initialize a dataset from a csv file. See documentation on *TabularDataset.from_df* for more details on arguments. :param csv_read_params: Keyword arguments to pass to the *pd.read_csv* method.

classmethod from_df(*df: pandas.core.frame.DataFrame, fields: Dict[Union[str, List[str]], Union[torchtable.field.core.Field, torchtable.field.core.FieldCollection, Collection[torchtable.field.core.Field]]], train=True*) → torchtable.dataset.core.TabularDataset

Initialize a dataset from a pandas dataframe.

Parameters

- **df** – pandas dataframe to initialize from

- **fields** – Dictionary mapping from a column identifier to a field or fields. The key can be a single column name or a tuple of multiple columns. The column(s) specified by the key will be passed to the field(s) transform method. The value can be a single field, a list/tuple of fields, or a *field.FieldCollection*. In general, each example in the dataset will mirror the structure of the fields passed. For instance, if you pass multiple fields for a certain key, the example will also have multiple outputs for the given key structured as a list. If you want a flat dictionary for the example, consider using the *flatten* attribute in the *field.FieldCollection* class (see *field.FieldCollection* documentation for more details).
- **train** – Whether this dataset is the training set. This affects whether the fields will fit the given data.

Example

```
>>> ds = TabularDataset.from_df(df, fields={
...     "authorized_flag": CategoricalField(handle_unk=False), # standard␣
↪field
...     "card_id": [CategoricalField(handle_unk=True),
...                 Field(LambdaOperator(lambda x: x.str[0]) > Categorize())],
↪ # multiple fields and custom fields
...     "price": NumericField(fill_missing=None, normalization=None, is_
↪target=True), # target field
...     ("authorized_flag", "price"): Field(LambdaOperator(
...             lambda x: (x["authorized_flag"] == "N").astype("int") * x[
↪"price"])), # multiple column field
... })
>>> ds[0]
{"authorized_flag": 0,
 "card_id": [1, 0],
  "price": 1.2,
  ("authorized_flag", "price"): 0.}
```

classmethod from_dfs(*train_df: pandas.core.frame.DataFrame, val_df: pandas.core.frame.DataFrame = None, test_df: pandas.core.frame.DataFrame = None, fields: Dict[Union[str, List[str]], Union[torchtable.field.core.Field, torchtable.field.core.FieldCollection, Collection[torchtable.field.core.Field]]] = None*) → Iterable[torchtable.dataset.core.TabularDataset]

Generates datasets from train, val, and test dataframes. .. rubric:: Example

```
>>> trn, val, test = TabularDataset.from_dfs(train_df, val_df=val_df, test_
↪df=test_df, fields={
...    "a": NumericField(), "b": CategoricalField(),
...  })
```

CHAPTER 5

torchtable.loader module

5.1 Module contents

class torchtable.loader.**DefaultLoader**(*dataset: torch.utils.data.dataset.Dataset*, *batch_size: int*, *device: Optional[torch.device] = None*, *repeat: bool = False*, *shuffle: Optional[bool] = None*)

Bases: torch.utils.data.dataloader.DataLoader

Defines an iterator that loads batches of data from a Dataset. Heavily based on the Iterator from torchtext.

Parameters

- **dataset** – The Dataset object to load examples from.
- **batch_size** – Batch size.
- **repeat** – Whether to repeat the iterator for multiple epochs.
- **shuffle** – Whether to shuffle examples between epochs.
- **device** (str or *torch.device*) – A string or instance of *torch.device* specifying which device the Variables are going to be created on. If None, the tensors will be created on cpu.

epoch

classmethod from_dataset(*dataset: torch.utils.data.dataset.Dataset*, *batch_size: int*, *device: torch.device = None*, *repeat: bool = False*, *shuffle: Optional[bool] = None*)

classmethod from_datasets(*train_ds: torch.utils.data.dataset.Dataset, batch_size: Union[T, Iterable[T]], val_ds: Optional[torch.utils.data.dataset.Dataset] = None, test_ds: Optional[torch.utils.data.dataset.Dataset] = None, device: Union[T, Iterable[T]] = None, repeat: Union[T, Iterable[T]] = False, shuffle: Optional[Union[T, Iterable[T]]] = None*) → Iterable[torchtable.loader.core.DefaultLoader]

init_epoch()

Set up the batch generator for a new epoch.

load_state_dict(*state_dict: Dict[str, Any]*)

state_dict() → Dict[str, Any]

CHAPTER 6

torchtable.model module

6.1 Module contents

class torchtable.model.**BatchHandlerModel**(*embs: List[torch.nn.modules.module.Module], batch_cat_field_getters: List[Callable[Dict, None._VariableFunctions.tensor]], batch_num_field_getters: Callable[Dict, None._VariableFunctions.tensor]*)

Bases: torch.nn.modules.module.Module

A model that takes a batch as input and converts it into a single 2-d tensor. Categorical fields are all embedded and the embeddings are all concatenated along with the numerical fields.

static field_to_embedding(*fld: torchtable.field.core.CategoricalField*) → torch.nn.modules.module.Module

forward(*batch*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

classmethod from_dataset(*dataset: torchtable.dataset.core.TabularDataset*) → DefaultModel

out_dim()

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index