Megatron Documentation

Release 0.1.0

Nash Taylor

Mar 05, 2019

Contents

CHAPTER 1

Pipelines

Pipelines are the core of Megatron. Pipelines contain all your transformations and are what you ultimately use to generate outputs.

class megatron.pipeline.**Pipeline**(*inputs*, *outputs*, *metrics=[]*, *explorers=[]*, *name=None*, *version=None*, *storage=None*, *overwrite=False*)

Bases: object

Holds the core computation graph that maps out Layers and manipulates data.

Parameters

- **inputs** (*list of megatron.Node(s)*) – input nodes of the Pipeline, where raw data is fed in.
- **outputs** (*list of megatron.Node(s)*) – output nodes of the Pipeline, the processed features.
- **name** (*str*) – unique identifying name of the Pipeline.
- **version** (*str*) – version tag for Pipeline's cache table in the database.
- **storage_db** (*Connection (defeault: 'sqlite')*) – database connection to be used for input and output data storage.

inputs
input nodes of the Pipeline, where raw data is fed in.

Type list of megatron.Node(s)

outputs
output nodes of the Pipeline, the processed features.

Type list of megatron.Node(s)

path
full topological sort of Pipeline from inputs to outputs.

Type list of megatron.Nodes

nodes
separation of Nodes by type.

Type dict of list of megatron.Node(s)

eager
when True, TransformationNode outputs are to be calculated on creation. This is indicated by data being passed to an InputNode node as a function call.

Type bool

name
unique identifying name of the Pipeline.

Type str

version
version tag for Pipeline's cache table in the database.

Type str

storage
storage database for input and output data.

Type Connection (defeault: None)

evaluate(*input_data*, *prune=True*)
Execute the metric Nodes in the Pipeline and get their results.

Parameters **input_data** (*dict of Numpy array*) – the input data to be passed to InputNodes to begin execution.

evaluate_generator(*input_generator*, *steps*)
Execute the metric Nodes in the Pipeline for each batch in a generator.

explore_generator(*input_generator*, *steps*)
Execute the explorer Nodes in the Pipeline for each batch in a generator.

fit(*input_data*, *epochs=1*)
Fit to input data and overwrite the metadata.

Parameters

- **input_data** (*2-tuple of dict of Numpy array, Numpy array*) – the input data to be passed to InputNodes to begin execution, and the index.
- **epochs** (*int (default: 1)*) – number of passes to perform over the data.

fit_generator(*input_generator*, *steps_per_epoch*, *epochs=1*)
Fit to generator of input data batches. Execute partial_fit to each batch.

Parameters

- **input_generator** (*generator of 2-tuple of dict of Numpy array and Numpy array*) – generator that produces features and labels for each batch of data.
- **steps_per_epoch** (*int*) – number of batches that are considered one full epoch.
- **epochs** (*int (default: 1)*) – number of passes to perform over the data.

partial_fit(*input_data*)
Fit to input data in an incremental way if possible.

Parameters **input_data** (*dict of Numpy array*) – the input data to be passed to InputNodes to begin execution.

save(*save_dir*)
Store the Pipeline and its learned metadata without the outputs on disk.

The filename will be {name of the pipeline}{version}.pkl.

Parameters **save_dir** (*str*) – the desired location of the stored nodes, without the filename.

transform(*input_data*, *index_field=None*, *prune=True*)
Execute the graph with some input data, get the output nodes' data.

Parameters

- **input_data** (*dict of Numpy array*) – the input data to be passed to InputNodes to begin execution.
- **index_field** (*str*) – name of key from input_data to be used as index for storage and lookup.
- **keep_data** (*bool*) – whether to keep data in non-output nodes after execution. activating this flag can be useful for debugging.

transform_generator(*input_generator*, *steps*, *index=None*)
Execute the graph with some input data from a generator, create generator.

Parameters

- **input_generator** (*dict of Numpy array*) – generator producing input data to be passed to Input nodes.
- **steps** (*int*) – number of batches to pull from input_generator before terminating.

megatron.pipeline.**load_pipeline**(*filepath*, *storage_db=None*)
Load a set of nodes from a given file, stored previously with Pipeline.save().

Parameters

- **filepath** (*str*) – the file from which to load a Pipeline.
- **storage_db** (*Connection (default: sqlite3.connect('megatron_default.db'))*) – database connection object to query for cached data from the Pipeline.

CHAPTER 2

Layers

Layers are how you build Pipelines. They're the transformations you're applying to your data.

2.1 Image Layers

These Layers are for transformations geared towards image data.

class megatron.layers.image.**Downsample**(*new_shape*)
Bases: megatron.layers.core.StatelessLayer

Shrink an image to a given size proportionally.

Parameters **new_shape** (*tuple of int*) – the target shape for the new image.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.image.**RGBtoBinary**(*keep_dim=True*)
Bases: megatron.layers.core.StatelessLayer

Convert image to binary mask where a 1 indicates a non-black cell.

Parameters **keep_dim** (*bool*) – if True, resulting image will stay 3D and will have 1 color channel. Otherwise 2D.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.image.**RGBtoGrey**(*method='luminosity'*, *keep_dim=False*)
Bases: megatron.layers.core.StatelessLayer

Convert an RGB array representation of an image to greyscale.

Parameters **method** (*{'luminosity', 'lightness', 'average'}*) –

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.image.**Upsample**(*new_shape*)
Bases: megatron.layers.core.StatelessLayer

Expand an image to a given size proportionally.

Parameters **new_shape** (*tuple of int*) – the target shape for the new image.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

2.2 Missing Data Layers

These Layers are for dealing with missing data.

class megatron.layers.missing.**Impute**(*imputation_dict*)
Bases: megatron.layers.core.StatelessLayer

Replace instances of one data item with another, such as missing or NaN with zero.

Parameters **imputation_dict** (*dict*) – keys of the dictionary are targets to be replaced; values are corresponding replacements.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

2.3 Numeric Layers

These Layers are for mathematical operations on your data, such as arithmetic.

class megatron.layers.numeric.**Add**
Bases: megatron.layers.core.StatelessLayer

Add up arrays element-wise.

transform(**arrays*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**Divide**(*impute=0*)
Bases: megatron.layers.core.StatelessLayer

Divide given array by another given array element-wise.

Parameters **impute** (*int/float or None*) – the value to impute when encountering a divide by zero.

transform(*X1*, *X2*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**Dot**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Multiply multiple arrays together as matrix multiplication.

transform(**arrays*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**ElementWiseMultiply**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Multiply two same-sized arrays element-by-element.

transform(*X*, *Y*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**Normalize**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Divide array by total to cause it to sum to one. If zero array, make uniform.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**ScalarMultiply**(*factor*)
Bases: megatron.layers.core.StatelessLayer

Multiply array by a given scalar.

Parameters **factor** (*float*) – multiplier.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**StaticDot**(*W*)
Bases: megatron.layers.core.StatelessLayer

Multiply array by a given matrix, as matrix mulitplication.

Parameters **W** (*np.array*) – matrix by which to multiply.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.numeric.**Subtract**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Subtract one array from another.

transform(*X1*, *X2*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

2.4 Shaping Layers

These Layers are for manipulating the shape of your data, from adding axes to creating time series windows.

class megatron.layers.shaping.**AddDim**(*axis=-1*)
Bases: megatron.layers.core.StatelessLayer

Add a dimension to an array.

Parameters **axis** (*int*) – the axis along which to place the new dimension.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Cast**(*new_type*)
Bases: megatron.layers.core.StatelessLayer

Re-defines the data type for a Numpy array's contents.

Parameters **new_type** (*type*) – the new type for the array to be cast to.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Concatenate**(*axis=-1*)
Bases: megatron.layers.core.StatelessLayer

Combine arrays along a given axis. Does not create a new axis, unless all 1D inputs.

Parameters **axis** (*int (default: -1)*) – axis along which to concatenate arrays. -1 means the last axis.

transform(**arrays*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Filter**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Apply given mask to given array along the first axis to filter out observations.

transform(*X*, *mask*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Flatten**(*n_outputs=1*, ***kwargs*)
Bases: megatron.layers.core.StatelessLayer

Reshape an array to be 1D.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**OneHotLabels**(*strict=False*)
Bases: megatron.layers.core.StatefulLayer

One-hot encode an array of categorical values, or non-consecutive numeric values.

partial_fit(*X*)
Update metadata based on given batch of data or full dataset.

Contains the main logic of fitting. This is what should be overwritten by all child classes.

Parameters **inputs** (*numpy.ndarray(s)*) – the input data to be fit to; could be one array or a list of arrays.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**OneHotRange**(*strict=False*)
Bases: megatron.layers.core.StatefulLayer

One-hot encode a numeric array where the values are a sequence.

partial_fit(*X*)
Update metadata based on given batch of data or full dataset.

Contains the main logic of fitting. This is what should be overwritten by all child classes.

Parameters **inputs** (*numpy.ndarray(s)*) – the input data to be fit to; could be one array or a list of arrays.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Reshape**(*new_shape*)
Bases: megatron.layers.core.StatelessLayer

Reshape an array to a given new shape.

Parameters **new_shape** (*tuple of int*) – desired new shape for array.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**Slice**(**slices*)
Bases: megatron.layers.core.StatelessLayer

Apply Numpy array slicing. Each slice corresponds to a dimension.

Slices (passed as hyperparameters) are constructed by the following procedure: - To get just N: provide the integer N as the slice - To slice from N to the end: provide a 1-tuple of the integer N, e.g. (5,). - To slice from M to N exclusive: provide a 2-tuple of the integers M and N, e.g. (3, 6). - To slice from M to N with skip P: provide a 3-tuple of the integers M, N, and P.

Parameters ***slices** (*int(s) or tuple(s)*) – the slices to be applied. Must not overlap. Formatting discussed above.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**SplitDict**(*fields*)
Bases: megatron.layers.core.StatelessLayer

Split dictionary data into separate nodes, with one node per key in the dictionary.

Parameters **fields** (*list of str*) – list of fields, dictionary keys, to be pulled out into their own nodes.

transform(*dicts*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

class megatron.layers.shaping.**TimeSeries**(*window_size*, *time_axis=1*, *reverse=False*)
Bases: megatron.layers.core.StatefulLayer

Adds a time dimension to a dataset by rolling a window over the data.

Parameters

- **window_size** (*int*) – length of the window; number of timesteps in the time series.
- **time_axis** (*int*) – on which axis in the array to place the time dimension.
- **reverse** (*bool (default: False)*) – if True, oldest data is first; if False, newest data is first.

partial_fit(*X*)
Update metadata based on given batch of data or full dataset.

Contains the main logic of fitting. This is what should be overwritten by all child classes.

Parameters **inputs** (*numpy.ndarray(s)*) – the input data to be fit to; could be one array or a list of arrays.

transform(*X*)
Apply transformation to given input data.

Parameters **inputs** (*np.ndarray(s)*) – input data to be transformed; could be one array or a list of arrays.

2.5 Text Layers

These Layers are for manipulating text data.

class megatron.layers.text.**RemoveStopwords**(*language='english'*)
Bases: megatron.layers.core.StatelessLayer

Remove common, low-information words from all elements of text array.

Parameters **language** (*str (default: english)*) – the language in which the text is written.

CHAPTER 3

Layer Wrappers

Wrappers allow you to use transformations from other modules as Layers in a Megatron Pipeline. Currently supported are:

- Keras models
- Sklearn transformations (including preprocessors, unsupervised models, supervised models)

CHAPTER 4

Metrics

Evaluation metrics attach to Nodes (usually a Model Node) and tell you how well that Node performed. Metrics are created just like other Layers, but they have their own behaviour.

class megatron.layers.metrics.**Metric**(*metric_fn*, ***kwargs*)

Bases: object

Layer type that holds an evaluation metric; only incorporated for Pipeline evaluation.

Parameters

- **metric_fn** (*function*) – the metric function to be wrapped.
- ****kwargs** – any keyword arguments to be passed to the metric when being called.

evaluate(**inputs*)

Run metric function on given input data.

CHAPTER 5

Reading and Writing Data (IO)

Megatron can currently read data from the following sources:

- Pandas Dataframes
- CSV files
- SQL database connections

When outputs have been calculated, they can be stored in association with their input observation index in a database. Any SQL database connection can be provided.

5.1 Datasets (Input)

megatron.io.dataset.**CSVData**(*filepath*, *exclude_cols=[]*, *nrows=None*)
Load fixed data from CSV filepath into Megatron Input nodes, one for each column.

Parameters

- **filepath** (*str*) – the CSV filepath to be loaded from.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.
- **nrows** (*int (default: None)*) – number of rows to load. If None, load all rows.

megatron.io.dataset.**PandasData**(*dataframe*, *exclude_cols=[]*, *nrows=None*)
Load fixed data from Pandas Dataframe into Megatron Input nodes, one for each column.

Parameters

- **dataframe** (*Pandas.DataFrame*) – the dataframe to be used.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.
- **nrows** (*int (default: None)*) – number of rows to load. If None, loads all rows.

megatron.io.dataset.**SQLData**(*connection*, *query*)
Load fixed data from SQL query into Megatron Input nodes, one for each column.

Parameters

- **connection** (*Connection*) – a database connection to any valid SQL database engine.
- **query** (*str*) – a valid SQL query according to the engine being used, that extracts the data for Inputs.

5.2 Data Generators (Input)

class megatron.io.generator.**CSVGenerator**(*filepath*, *batch_size=32*, *exclude_cols=[]*)
Bases: object

A generator of data batches from a CSV file in pipeline Input format.

Parameters

- **filepath** (*str*) – the CSV filepath to be loaded from.
- **batch_size** (*int*) – number of observations to yield in each iteration.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.

class megatron.io.generator.**PandasGenerator**(*dataframe*, *batch_size=32*, *exclude_cols=[]*)
Bases: object

A generator of data batches from a Pandas Dataframe into Megatron Input nodes.

Parameters

- **dataframe** (*Pandas.DataFrame*) – dataframe to load data from.
- **batch_size** (*int*) – number of observations to yield in each iteration.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.

class megatron.io.generator.**SQLGenerator**(*connection*, *query*, *batch_size=32*, *limit=None*)
Bases: object

A generator of data batches from a SQL query in pipeline Input format.

Parameters

- **connection** (*Connection*) – a database connection to any valid SQL database engine.
- **query** (*str*) – a valid SQL query according to the engine being used, that extracts the data for Inputs.
- **batch_size** (*int*) – number of observations to yield in each iteration.
- **limit** (*int*) – number of observations to use from the query in total.

5.3 Storage (Output)

class megatron.io.storage.**DataStore**(*table_name*, *version*, *db_conn*, *overwrite*)
Bases: object

SQL table of input data and output features, associated with a single pipeline.

Parameters

- **table_name** (*str*) – name of pipeline's cache table in the database.
- **version** (*str*) – version tag for pipeline's cache table in the database.
- **db_conn** (*Connection*) – database connection to query.

read(*cols=None*, *rows=None*)

Retrieve all processed features from cache, or lookup a single observation.

For features that are multi-dimensional, use pickle to read string.

Parameters

- **cols** (*list of int (default: None)*) – indices of output columns to retrieve. If None, get all columns.
- **rows** (*list of any or any (default: None)*) – index value to lookup output for, in dictionary form. If None, get all rows. should be the same data type as the index.

write(*output_data*, *data_index*)

Write set of observations to database.

For features that are multi-dimensional, use pickle to compress to string.

Parameters

- **output_data** (*dict of ndarray*) – resulting features from applying pipeline to input_data.
- **data_index** (*np.array*) – index of observations.

CHAPTER 6

Nodes

6.1 Core Nodes

Nodes are the internal building blocks of Pipelines. While you're usually not using them directly, it's helpful to understand how they work.

class megatron.nodes.core.**InputNode**(*name*, *shape=()*)

Bases: *megatron.nodes.core.Node*

A pipeline node holding input data as a Numpy array.

It is always an initial node in a Pipeline (has no inbound nodes) and, when run, stores its given data (either from a feed dict or a function call) in its output.

Parameters

- **name** (*str*) – a name to associate with the data; the keys of the Pipeline feed dict will be these names.
- **shape** (*tuple of int*) – the shape, not including the observation dimension (1st), of the Numpy arrays to be input.

name

a name to associate with the data; the keys of the Pipeline feed dict will be these names.

Type str

shape

the shape, not including the observation dimension (1st), of the Numpy arrays to be input.

Type tuple of int

load(*observations*)

Validate and store the data passed in.

Parameters **observations** (*np.ndarray*) – data from either the feed dict or the function call, to be validated.

Raises `megatron.utils.ShapeError` – error indicating that the shape of the data does not match the shape of the node.

validate_input(*observations*)
Ensure shape of data passed in aligns with shape of the node.

Parameters **observations** (*np.ndarray*) – data from either the feed dict or the function call, to be validated.

Raises `megatron.utils.ShapeError` – error indicating that the shape of the data does not match the shape of the node.

class megatron.nodes.core.**Node**(*inbound_nodes*)
Bases: `object`

Base class of pipeline nodes.

Parameters **inbound_nodes** (*list of megatron.Node*) – nodes who are to be connected as inputs to this node.

inbound_nodes
nodes who are to be connected as inputs to this node.

Type list of megatron.Node

outbound_nodes
nodes to whom this node is connected as an input.

Type list of megatron.Node

output
holds the data output by the node's having been run on its inputs.

Type np.ndarray

outbounds_run
number of outbound nodes that have been executed. this is a helper for efficiently removing unneeded data.

Type int

traverse(**path*)
Return a Node from elsewhere in the graph by navigating to it from this Node.

A negative number indicates moving up to a parent, a positive number down to a child. The number itself is a 1-based index into the parents/children, from left to right. For example, a step of -2 will go to the second parent, while a step of 3 will go to the third child.

Parameters **path** (*ints) – Arbitrary number of integers indicating the steps in the path.

Returns the node at the end of the provided path.

Return type *Node*

class megatron.nodes.core.**TransformationNode**(*layer*, *inbound_nodes*, *layer_out_index=0*)
Bases: *megatron.nodes.core.Node*

A pipeline node holding a Transformation.

It connects to a set of input Nodes (of class Node or Input) and, when run, applies its given Transformation, storing the result in its output variable.

Parameters

- **layer** (*megatron.Layer*) – the Layer to be applied to the data from its inbound Nodes.

- **inbound_nodes** (*list of megatron.Node / megatron.Input*) – the Nodes to be connected to this node as input.
- **layer_out_index** (*int (default: 0)*) – when a Layer has multiple return values, shows which one corresponds to this node.

transformation
the transformation to be applied to the data from its input Nodes.

Type megatron.Transformation

output
is None until Node is run; when run, the Numpy array produced is stored here.

Type None or np.ndarray

is_fitted
indicates whether the Transformation inside the Node has, if necessary, been fit to data.

Type bool

fit()
Apply fit method from Layer to inbound Nodes' data.

partial_fit()
Apply partial fit method from Layer to inbound Nodes' data.

transform(*prune=True*)
Apply and store result of transform method from Layer on inbound Nodes' data.

Parameters **prune** (*bool (default: True)*) – whether to erase data from intermediate nodes after they are fully used.

6.2 Loading Nodes From Files

A set of Nodes can be defined according to the schema of a given data source. Here's how.

megatron.nodes.fromfile.**from_csv**(*filepath*, *exclude_cols=[]*, *eager=False*, *nrows=None*)
Load Input nodes from columns of a CSV file.

Parameters

- **filepath** (*str*) – path of CSV file to be loaded.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.
- **eager** (*bool*) – whether to load data as well, making for eager execution.
- **nrows** (*int*) – number of rows to load when eager is True. Default is for all rows to load.

megatron.nodes.fromfile.**from_dataframe**(*df*, *exclude_cols=[]*, *eager=False*, *nrows=None*)
Load Input nodes from columns of a Pandas dataframe.

Parameters

- **df** (*Pandas.DataFrame*) – dataframe from which to load columns.
- **exclude_cols** (*list of str (default: [])*) – any columns that should not be loaded as Input.
- **eager** (*bool*) – whether to load data as well, making for eager execution.
- **nrows** (*int*) – number of rows to load when eager is True. Default is for all rows to load.

megatron.nodes.fromfile.**from_sql**(*connection*, *query*, *eager=False*, *nrows=None*)

Load Input nodes from columns of a Pandas dataframe.

Parameters

- **connection** (*Connection*) – database connection to load from.
- **query** (*str*) – query to execute in connection to load columns.
- **eager** (*bool*) – whether to load data as well, making for eager execution.
- **nrows** (*int*) – number of rows to load when eager is True. Default is for all rows to load.

Megatron is a Python module for building data pipelines that encapsulate the entire machine learning process, from raw data to predictions.

The advantages of using Megatron:

- A wide array of data transformations can be applied, including:
  - Built-in preprocessing transformations such as one-hot encoding, whitening, time-series windowing, etc.
  - Any custom transformations you want, provided they take in Numpy arrays and output Numpy arrays.
  - Sklearn preprocessors, unsupervised models (e.g. PCA), and supervised models. Basically, anything from sklearn.
  - Keras models.
- To any Keras users, the API will be familiar: Megatron's API is heavily inspired by the Keras Functional API, where each data transformation (whether a simple one-hot encoding or an entire neural network) is applied as a Layer.
- Since all datasets should be versioned, Megatron allows you to name and version your pipelines and associated output data.
- Pipeline outputs can be cached and looked up easily for each pipeline and version.
- The pipeline can be elegantly visualized as a graph, showing connections between layers similar to a Keras visualization.
- Data and input layer shapes can be loaded from structured data sources including:
  - Pandas dataframes.
  - CSVs.
  - SQL database connections and queries.
- Pipelines can either take in and produce full datasets, or take in and produce batch generators, for maximum flexibility.
- Pipelines support eager execution for immediate examination of data and simpler debugging.

CHAPTER 7

Installation

To install megatron, just grab it from pip:

```
pip install megatron
```

There's also a Docker image available with all dependencies and optional dependencies installed:

```
docker pull ntaylor22/megatron
```

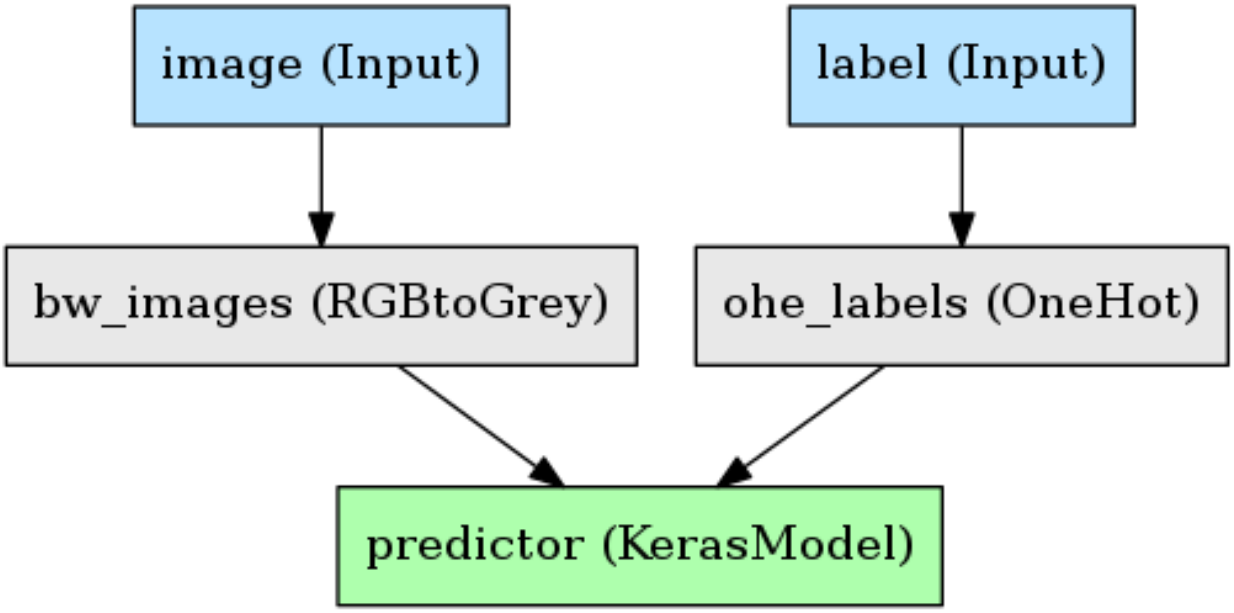
7.1 Optional Dependencies

- Scikit-Learn
  - If you'd like to use Sklearn transformations as Layers.
- Keras
  - If you'd like to use Keras models as Layers.
- Pydot
  - If you'd like to be able to visualize pipelines.
  - Note: requires GraphViz to run.

CHAPTER 8

Tutorial

Let's build the following pipeline:



A simple example with an image and some binary labels. The goals here are:

- Convert the RGB image to black and white.
- One-hot encode the labels.
- Feed these into a Keras CNN, get predictions.

Let's assume a built and compiled Keras model named "model" has already been made:

```
model = Model(image_input, two_class_output)
```

Let's start by making the input nodes for the pipeline:

```
images = megatron.nodes.Input('image', (48, 48, 1))
labels = megatron.nodes.Input('label')
```

The first argument is a name, which is a mandatory argument for Inputs.

As for the second argument, by default, the shape of an Input is a 1D array, so we don't need to specify the shape of 'label', but we will for 'image', which has a particular shape. The shape does _not_ include the first dimension, which is the observations.

Now let's apply greyscaling to the image, and one-hot encoding to the labels:

```
grey_images = megatron.layers.RGBtoGrey(method='luminosity', keep_dim=True)(images)
ohe_labels = megatron.layers.OneHotRange(max_val=1)(labels)
```

4 things to note here:

- Calling a Layer produces a Node. That means Layers can be re-used to produce as many Nodes as we want, though we're not taking advantage of that here.
- The initialization arguments to a Layer are its "hyperparameters", or configuration parameters, such as the method used for converting RGB to greyscale.
- The first argument when calling a Layer is the previous layers you want to call it on. If there's multiple inputs, they should be provided as a list.
- The second argument is the name for the resulting node. If this node is to be an output of the model, it must be named; otherwise, names are not necessary, though still helpful for documentation.

With our features and labels appropriately processed, we can pass them into our Keras model:

```
preds = megatron.layers.Keras(model)([grey_images, ohe_labels])
```

Since this is an output of the pipeline, we name it. Lastly, let's attach a metric to the Keras model so we know how well it did:

```
acc = megatron.metrics.Accuracy()([ohe_labels, preds])
```

Note that metrics do not behave like other layers; they are not executed when we fit or transform. They come into play if we evaluate a Pipeline, at which point all the pipeline's metrics will be calculated and given back to us. We'll see that in a second.

To be able to identify the different metrics, it's required that we name them, as the second argument to the call.

Finally, let's create the pipeline by defining its inputs and outputs, just like a Keras Model:

```
storage_db = sqlite3.connect('getting_started')
pipeline = megatron.Pipeline([images, labels], preds, name='getting_started',␣
↪version=0.1, storage=storage_db)
```

Let's break down the arguments here:

- The first argument is either a single node or a list of nodes that are meant to be input nodes; that is, they will have data passed to them.
- The second argument is either a single node or a list of nodes that are meant to be output nodes; that is, when we run the pipeline, they're the nodes whose data we'll get.
- The pipeline must be named, and it can have a version number, but that is optional. These identifiers will be used for caching processed data and the pipeline itself.

- You can store the output data of a pipeline in a SQL database, and look it up using the index of the observations. If no index is provided (we provided no index here), it's simply integers starting from 0.

Now let's train the model, get the predictions, then lookup the prediction for the first observation from the storage database:

```
data = {'images': np.random.random((1000, 48, 48, 3)),
        'labels': np.random.randint(0, 1, 1000)}
pipeline.fit(data)
outputs = pipeline.transform(data)
one_output = pipeline.storage.read(rows=['0'])
print(outputs[0].shape) # --> (1000, 2)
print(one_output[0].shape) # --> (1, 2)

metrics = pipeline.evaluate(data)
print(metrics['model_acc']) # --> 0.51
```

What did we learn here?

- We pass in data by creating a dictionary, where the keys are the names of the input nodes of the pipeline, and the values are the Numpy arrays.
- Calling .transform(data) gives us a dictionary, where the keys are the names of the output nodes of the pipeline, and the values are the Numpy arrays.
- Looking up observations by index in the storage database gives us a dictionary with the same structure as .transform(data).
- Metrics are calculated by calling .evaluate(data) on the pipeline.

Finally, let's save the pipeline to disk so it can be reloaded with its structure and trained parameters. Let's save it under the directory "pipelines/", from the current working directory:

```
pipeline.save('pipelines/')
```

The pipeline has been saved at the following location: [working_directory]/pipelines/getting_started-0.1.pkl. The name of the pickle file is the name of the pipeline and the version number, defined in its initialization, separated by a hyphen.

Let's reload that pipeline:

```
pipeline = megatron.load_pipeline('pipelines/getting_started-0.1.pkl', storage_
↪db=storage_db)
```

We provide the filepath for the pipeline we want to reload, and one extra argument: since we can't pickle database connections, when we want to connect to the storage database, we have to make that connection variable and pass it as the second argument to load_pipeline. If you aren't using caching, you don't need to do this.

To summarize:

- We created a Keras model and some data transformations.
- We connected them up as a pipeline, ran some data through that pipeline, and got the results.
- We stored the results and the fitted pipeline on disk, looked up those results from disk, and reloaded the pipeline from disk.
- The data and pipeline were named and versioned, and the observations in the data had an index we could use for lookup.

CHAPTER 9

Custom Layers

If you have a function that takes in Numpy arrays and produces Numpy arrays, you have two possible paths to adding it as a Layer in a Pipeline:

1. The function has no parameters to learn, and will always return the same output for a given input. We refer to this as a "stateless" Layer.
2. The function learns parameters (i.e. needs to be "fit"). We refer to this as a "stateful" Layer.

9.1 Custom Stateful Layers

To create a custom stateful layer, you will inherit the StatefulLayer base class, and write two methods: fit (or partial_fit), and transform. Here's an example with a Whitening Layer:

```
class Whiten(megatron.layers.StatefulLayer):
    def fit(self, X):
        self.metadata['mean'] = X.mean(axis=0)
        self.metadata['std'] = X.std(axis=0)

    def transform(self, X):
        return (X - self.metadata['mean']) / self.metadata['std']
```

There's a couple things to know here:

- When you calculate parameters during the fit, you store them in the provided dictionary self.metadata. You then retrieve them from this dictionary in your transform method.
- If your Layer is one that can be fit iteratively, you can override partial_fit rather than fit. If your transformation cannot be fit iteratively, you override fit; note that Layers without a partial_fit cannot be used with data generators, and will throw an error in that situation.
  - For an example of how to write a partial_fit method, see megatron.layers.shaping.OneHotRange.).

9.2 Custom Stateless Layers

To create a custom stateless Layer, you can simply define your function and wrap it in megatron.layers.Lambda. For example:

```
def dot_product(X, Y):
    return np.dot(X, Y)

dot_xy = megatron.layers.Lambda(dot_product)([X_node, Y_node], 'dot_product_result')
```

That's it, a simple wrapper.

CHAPTER 10

Why is it called Megatron?

Because the layers are data transformers!

That's… that's about it.

CHAPTER 11

License

MIT.

Python Module Index

m

Index

P

R

S

T

U

V

W