paragraph

Feb 14, 2020

Documentation

A pure Python micro-framework supporting seamless lazy and concurrent evaluation of computation graphs.

CHAPTER 1

Introduction

Paragraph adds the *functional programming paradigm* to Python in a minimal fashion. One additional class, `Variable`, and a function decorator, `op`, is all it takes to turn regular Python code into a *computation graph*, i.e. a computer representation of a system of equations:

```
>>> import paragraph as pg
>>> import operator
>>> x, y = pg.Variable("x"), pg.Variable("y")
>>> add = pg.op(operator.add)
>>> s = add.op(x, y)
```

The few lines above fully instantiate a computation graph, here in its simplest form with just one equation relating `x`, `y` and `s` via the function `add`. Given values for the input variables `x` and `y`, the value of `s` is resolved as follows:

```
>>> pg.evaluate([s], {x: 5, y: 10})
[15]
```

CHAPTER 2

Key features

The main benefits of using paragraph stem from the following features of `pg.session.evaluate`:

Lazy evaluation Irrespective of the size of the computation graph, only the operations required to evaluate the output variables are executed. Consider the following extension of the above graph:

```
>>> z = pg.Variable("z")
>>> t = add.op(y, z)
```

Then the statement:

```
>>> pg.evaluate([t], {y: 10, z: 50})
[60]
```

just ignores the variables `s` and `x` altogether, since they do not contribute to the evaluation of `t`. In particular, the operation `add(x, y)` is not executed.

Eager currying Invoking an op with invariable arguments (that is, arguments that are not of type `Variable`) just returns an invariable value: evaluation is eager whenever possible. If invariable arguments are provided for a subset of the input variables, the computation graph can be simplified using `solve`, which returns a new variable:

```
>>> u_x = pg.solve([u], {y: 10, z: 50})[0]
```

Here, `u_x` is a different variable from `u`: it now depends on a single input variable (`x`), and it knows nothing about a variable `y` or `z`, instead storing a reference to the value of their sum `t`, i.e. `60`.

Thus, `pg.session.solve` acts much as `functools.partial`, except it simplifies the system of equations where possible by executing dependent operations whose arguments are invariable.

Graph composition Assume a variable `y` depends on a number of input variables `x_1`,..., `x_p`, and another variable `v` on `u_1`,...,‘‘u_q‘‘ (not necessarily different), and `v` should be identified to `x_p`. The following statement:

```
>>> y_v = pg.solve([y], args={x_p: v})[0]
```

returns a new variable `y_v` that depends on `x_1`,..., `x_{p-1}` as well as on `u_1`,..., `u_q`, as if the two computation graphs defining `y` and `v` had been pieced together.

Note that the respective input variables may overlap, with the restriction that `v` should not depend on `x_p` as that would result in a circular dependency. Also, additional arguments may be added to `args` in the statement above to set further values of the input variables `x_1,...,x_{p-1}`. However, values cannot be set for `u_1,...,u_q` here, since they are not dependencies of `y`, but of `y_v`.

Transparent multithreading Invoking `evaluate` or `solve` with an instance of `concurrent.ThreadPoolExecutor` will allow independent blocks of the computation graph to run in separate threads:

```
>>> with ThreadPoolExecutor as ex:
...     res = pg.evaluate([z_t], {t: 5}, ex)
```

This is particularly beneficial if large subsets of the graph are independent.

CHAPTER 3

Constraints

3.1 Side-effects

The features listed above come at some price, essentially because the order in which operations are actually executed generally differs from the order of their invocations. For paragraph to guarantee that a variable always evaluates to the same value given the same inputs, as in a system of mathematical equations, it is paramount that operations remain free of side-effects, i.e. they **never** mutate an object they received as an argument, or store as an attribute. The state sequence of the object would be, by definition, out of the control of the programmer.

There is close to nothing paragraph can do to prevent such a thing happening. When in doubt, make sure to operate on a copy of the argument.

3.2 Typing

Variables do not carry any information regarding the type of the value they represent, which precludes binding a method of the underlying value to an instance of `Variable`: such instructions can appear only within the code of an op. Since binary operators are implemented using special methods in Python, this also precludes such statements as:

```
>>> s = x + y
```

for this would be resolved by the Python interpreter into `s = x.__add__(y)`, then `s = y.__radd__(x)`, yet none of these methods is defined by `Variable`.

For more information please consult the documentation.

CHAPTER 4

Going further

4.1 Partial evaluation

When the arguments passed to *paragraph.session.evaluate* are insufficient to resolve fully an output variable (that is, at least one transitive dependency of the output variable is left uninitialized), the value returned for the output variable is simply another variable. This new variable has in general fewer dependencies, for dependencies fully resolved upon evaluation are replaced by their values.

Note: The ambiguity on the type returned by `paragraph.session.evaluate` will be lifted in version 2.0. From there on, `paragraph.session.evaluate` will raise in situations such as described above. The support for partial evaluation will however be continued using the new function `paragraph.session.solve`.

4.2 Mapping over inputs

The function *paragraph.session.apply* extends *paragraph.session.evaluate* to take, in addition, an iterator over input arguments to the computation graph. It takes advantage of partial evaluation to reduce the number of operations evaluated at each iteration.

4.3 Concurrency

Building upon the guarantees granted by a forward traversal, concurrent execution of ops comes at no additional cost. This feature relies on the *concurrent* package from the Python standard library: ops are simply submitted to an instance of *concurrent.futures.Executor* for evaluation. The executor should be provided externally and allow calling *concurrent.futures.Future* methods from a submitted callable (in particular, this excludes *concurrent.futures.ProcessPoolExecutor*). The responsibility for shutting down the executor properly lies on the user. In absence of an executor, variables are evaluated in a sequential manner, yet still lazily.

Should an operation be executed in the main process, it can be marked as such by setting the attribute *Op.thread_safe* to False.

Example usage:

```
>>> ...graph definition...
>>> with ThreadPoolExecutor() as ex:
...     res = evaluate([output], args={input: input_value}, executor=ex)
```

Note: Argument values passed to *paragraph.session.evaluate* can be of type *concurrent.futures.Future*, in which case the consuming operations will simply block until the result is available.

Note: Similarly, an executor can be passed to the function *paragraph.session.apply*.

4.4 Eager mode

Within the context manager *paragraph.session.eager_mode*, ops are executed eagerly: the underlying *_run* method is invoked directly rather than returning an instance of *Variable*. In this mode, arguments of type *Variable* are generally not accepted. No concurrent evaluation occurs in eager mode.

This mode is particularly useful when testing or debugging a computation graph without modifying the code defining it, by simply bypassing the machinery set up by the framework.

4.5 Backward propagation

Conversely, information can be backward propagated through the computation graph using *Requirements*. Where applicable, an op can implement the *arg_requirements* method that resolves the requirement bearing on each of its arguments given this bearing on its ouput. This comes in handy e.g. when a particular time range should be available from the output, while rolling operations (such as sum, average,...) are performed in the graph (or any operation requiring a additional "prefetch" operations from the past).

The *arg_requirements* method receives the requirements bearing on the output variable and the name of a variable argument of the operation, and returns the requirements that should bear on the said variable argument.

Requirements are substantiated by mixin classes, which add attributes and assume full responsibility for their proper aggregation. They are usually defined in the same module as the operations using them. Then, a *compound requirements* class is simply defined by:

```
>>> @attr.s
... class MyRequirements(DateRangeRequirement, DatasetContentsRequirement):
...     pass
```

A requirement class must define the method *merge(self, other)* that aggregates requirements (more accurately, the requirement attributes it defines) arising from multiple usages of the same variable. This method should fulfill a small number of properties documented in the base class.

Once all components are in place, requirements can be backpropagated:

```
>>> reqs = solve_requirements(output=v2, output_requirements=MyRequirements(date_
↪range=ExactRange("2001-01-01", "2001-02-01")))
>>> reqs[v1].date_range  # Holds the backpropagated required date_range
```

CHAPTER 5

Caveats

5.1 Side effects

The order in which variables are evaluated should not be expected to match the order in which they are defined. As a consequence, it is *not safe* for operations to change variable arguments *in place* (aka side effects). As Python offers no mechanism to prevent side-effects, it is the responsibility of the user to ensure that copies are returned instead.

For the very same reasons, operations and graphs should be stateless, as their state sequence would otherwise lie outside of the control of the author of a computation graph.

CHAPTER 6

Glossary

variable Throughout this module, the term _variable_ should be understood in its mathematical sense. A variable can be unbound, and serve as an input placeholder, or bound, and symbolize the result of a certain operation applied to a certain set of arguments, at least one of which is also a variable.

operation An operation (or simply op) relates variables together.

transitive dependency A dependency of a variable is any other variable related to it by an operation. The *transitive* dependencies of a variable are the variables whose values enter its own evaluation, i.e. all variables in the union of its dependencies, their own dependencies, and so on until no more dependency is found. Together with the initial dependent variable, they form the *computation graph spanned* by the latter.

boundary A boundary is an arbitrary list of variables whose dependencies are excluded from the transitive dependency. The set of unbound variables is a canonical boundary associated to the transitive dependencies of all its variables. In the context of this module, it essentially allows to prune computation branches whose evaluation is not required.

traversal An ordering of the variables resulting from following the dependency relationships (the edges) of a computation graph. Dependency relationships can be excluded by setting a boundary to the traversal.

forward traversal Depth-first *traversal* of a computation graph, where every dependent variable occurs after all its dependencies. In this order, variables can be evaluated in turn, as the values of their dependencies are resolved before their own resolution occurs.

backward traversal Breadth-first *traversal* of a computation graph, where a dependency occurs after all the variables depending on it, directly or transitively. In this order, information can be backward propagated through the graph.

CHAPTER 7

Session

Algorithms for traversing, solving and evaluating computation graphs

paragraph.session.**eager_mode**()
Activate eager mode within a context manager.

In eager mode, the method `Op.op` is replaced with the direct invocation of the underlying method `Op._run`. In this mode, no variable is emitted, allowing to test a computation graph without ever calling `session.evaluate`.

paragraph.session.**traverse_fw**(*output*)
Returns a generator implementing a *forward traversal* of the computation subgraph leading to *var*.

The generator returned guarantees that every dependent variable occurs after all its dependencies upon iterating, whence the name *forward traversal*. When generated in this order, variables can be simply evaluated in turn: at each iteration, all dependencies of the current variable will have been evaluated already.

Parameters **output** (Iterable[*Variable*]) – The variables whose dependencies should be traversed.

Yields All dependencies of the output variables, each variable yielded occurring before the variables depending thereupon.

Raises ValueError – If a cyclic dependency is detected in the graph.

Return type Generator[*Variable*, None, None]

paragraph.session.**evaluate**(*output*, *args*, *executor=None*)
Evaluate the specified output variable.

The argument values provided through *args* should be:

- of the type expected by the operations consuming the variable,
- of type `concurrent.futures.Future`, in which case the result will be awaited by consuming ops. The result should be of the expected type.

Support of arguments values of type Variable will be dropped in version 2.0 and a DeprecationWarning will be issued. The same applies if any input variable required to evaluate the output is left uninitialized.

Parameters

- **output** (Iterable[*Variable*]) – The variables to evaluate.
- **args** (Dict[*Variable*, Any]) – Initialization of the input variables, none of which should have dependencies.
- **executor** (Optional[Executor]) – An instance of concurrent.futures.Executor, to which op evaluations are submitted. If None, the default, evaluation proceeds sequentially.

Return type List

Returns A list of values of the same size as *output*. The entry at index *i* is the computed value of *output[i]*.

Raises ValueError – If a variable in *args* is not an input variable. In this case, the consistency of the results cannot be guaranteed.

paragraph.session.**solve**(*output*, *args*, *executor=None*)

Resolve the specified output variables.

The argument values provided through *args* should be:

- of the type expected by the operations consuming the variable,
- of type Variable, in which case it should evaluate to the above type,
- of type concurrent.futures.Future, in which case the result will be awaited by consuming ops. The result should be of either above types.

Parameters

- **output** (Iterable[*Variable*]) – The variables to evaluate.
- **args** (Dict[*Variable*, Any]) – Initialization of the input variables, none of which should have dependencies.
- **executor** (Optional[Executor]) – An instance of concurrent.futures.Executor, to which op evaluations are submitted. If None, the default, evaluation proceeds sequentially.

Return type List

Returns A list of variables of the same size as *output*. The entry at index *i* is the resolved variable for *output[i]*.

Raises ValueError – If a variable in *args* is not an input variable. In this case, the consistency of the results cannot be guaranteed.

paragraph.session.**apply**(*output*, *args*, *iter_args*, *executor=None*)

Iterate the evaluation of a set of output variables over input arguments.

This function accepts two types of arguments: *args* receives *static* arguments, using which a first evaluation of the output variables is executed; then, *iter_args* receives an iterable over input arguments, which are iterated over to resolve the output variables left unresolved after the first evaluation. The values of the output variables obtained after each iteration are then yielded. See *evaluate()* for the constraints bearing on the types of the values provided in both *args* and *iter_args*.

Parameters

- **output** (List[*Variable*]) – The variables to evaluate.
- **args** (Dict[*Variable*, Any]) – A dictionary mapping input variables onto input values.
- **iter_args** (Iterable[Dict[*Variable*, Any]]) – An iterable over dictionaries mapping input variables onto input values.

- **executor** (Optional[Executor]) – An instance of concurrent.futures.Executor to which op evaluations are submitted. If None (the default), evaluation proceeds sequentially.

Yields A list of values of the same size as *output*. The entry at index *i* is the computed value of *output[i]*.

Raises ValueError – If a dynamic argument assigns a value to a variable appearing in static arguments, as proceeding would produce inconsistent results.

Return type Generator[List[Any], None, None]

paragraph.session.**traverse_bw**(*output*)

Returns a generator implementing a *backward traversal* of *var*'s transitive dependencies.

This generator guarantees that a variable is yielded after all its usages.

Note: If *var* is in the boundary, the generator exits without yielding any variable.

Parameters **output** (List[*Variable*]) – The variables whose transitive dependencies should be explored.

Yields All dependencies of the output variables (stopping at the boundary), each variable is yielded after all variables depending thereupon.

Return type Generator[*Variable*, None, None]

paragraph.session.**solve_requirements**(*output_requirements*)

Backward propagate requirements from the output variables to their transitive dependencies

Parameters **output_requirements** (Dict[*Variable*, *Requirement*]) – the requirements to be fulfilled on output

Return type Dict[*Variable*, *Requirement*]

Returns A dictionary mapping all transitive dependencies of *output* onto their resolved requirement dictionaries

CHAPTER 8

Classes and decorator

Class definitions supporting the computation graphs.

class paragraph.types.**Variable**(*name=None*, *op=None*, *args=NOTHING*, *dependencies=NOTHING*)

Bases: `object`

A generic *variable*.

This class is the return type of all operations in a computation graph. Independent variables can be instantiated directly as follows:

```
>>> my_var = Variable(name="my_var")
```

Note: Type annotations are missing for attributes *op* and *args*, for Sphinx does not seem to cope with forward references.

name
a string used to represent the variable. The attribute is mandatory for independent variables and None (the default) otherwise.

op
the operation producing the variable, of type *Op*.

args
a dictionary mapping arguments of the above *op* onto their values, of type *Dict[Variable, Any]*.

dependencies
a dictionary mapping arguments of the above callable onto other variables, of type *Dict[str, Variable]*.

isinput()

Return type `bool`

isdependent()

Return type `bool`

class paragraph.types.**Requirement**
Bases: abc.ABC

Base class for defining a requirement mixin

A Requirement class defines one or several attributes storing the actual requirement being expressed, and takes complete responsibility of these attributes. Therefore, a concrete requirement class should:

- define default values or factory callbacks for *all* the attributes it introduces,
- redefine the *merge()* method below appropriately, respecting the prescribed constraints,
- expose convenience methods to manipulate their attributes, these methods should operate *in place* to avoid unwanted interference with other Requirement mixins,
- provide an implementation of __deepcopy__() whenever relevant .

Deriving a compound requirements class boils down to putting together requirement mixins:

```
>>> @attr.s
>>> class MyRequirements(DateRangeRequirement, DatasetContentsRequirement):
...     pass
```

assuming *DateRangeRequirement* and *DatasetContentsRequirement* derive from the present class.

merge(*other*)
Merges *other* into *self* in-place, must be implemented by all cooperating mixin classes.

The requirement mixins should define the *merge()* method for the attributes they define (and therefore take responsibility for). The computation implemented must be both:

commutative the result of *[self.merge(usg) for usg in usages[self]]* does not depend on the order of the usages in the list

idempotent if *usg1 = usg2*, invoking *self.update(usg2)* after *self.update(usg1)* leaves *self* unchanged

Only under these conditions is it guaranteed that all *backward traversals* of the computation graph result in the same input requirements.

In addition, every implementation of this method must be *cooperative* and end with the following line:

```
super().merge(other)
```

Parameters **other** – the requirement to merge into self, must be of the same concrete type as *self*.

new()
Return an empty requirement of the same type as self.

class paragraph.types.**Op**(*, *thread_safe=True*)
Bases: object

Class of all computation graph operations.

A concrete Op class should redefine the _run() method, which fully specifies its behavior. Calling an Operation instance results in the following:

- if no argument is of Variable type, the return value of the _run() method is returned,
- otherwise, a Variable is returned, which represents the result of the operation applied to its arguments.

Additional requirements can be introduced for variable arguments, globally or individually, by redefining the _arg_requirements() method appropriately.

thread_safe
If False, the op is always executed in the main thread. Defaults to True.

static split_args(*args*)
Separate positional from keyword arguments in a dict.

This method extracts *args* entries with a key of type integer, and collates them into a list

Return type `Tuple[List[Any], Dict[str, Any]]`

arg_requirements(*req*, *arg=None*)
Compute the requirements on the input value for argument *arg* from the requirements *req* bearing on the output variable.

The base implementation below returns an empty requirement for every argument, preserving the type. Concrete classes should redefine this method whenever applicable.

> **Warning:** This method should never update requirements *in-place*, as this would result in adversary side-effects. The recommended practice is for this method to: - instantiate a new Requirement instance or deep-copy the passed in *req* if information should be retained, - update the new instance using in-place methods exposed by the requirement class, and return it.
>
> Should the output requirement be returned unmodified, it is however safe to just return its reference as-is.

Return type *Requirement*

op(**args*, ***kwargs*)
Define a variable as the result of applying the op to the arguments provided.

While this method always returns a Variable instance, it accepts:

- _concrete_ arguments of the type expected by `_run` at the same position/for the same keyword,
- Variable instances that resolve to a value of the expected type.

Return type *Variable*

`paragraph.types.`**op**(*func*)
Wraps a function within an Op object.

The returned function accepts arguments of type Variable everywhere in its signature, in addition to the types accepted by the decorated function. In presence of such arguments, it returns a Variable object symbolizing the result of the operation with the arguments passed in. In absence of variable arguments, the function returned is just equivalent to the function passed in, in particular it returns a value of the pristine return type.

> **Warning:** Operations returned by this decorator are marked thread-safe by default. It is the user's responsibility to set *Op.thread_safe* to *False* where appropriate.

Parameters **func** (`Callable`) – the function to transform into an Op

Return type *Op*

CHAPTER 9

Versions

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

All release versions should be documented here with release date and types of changes. Unreleased changes and pre-releases (i.e. alpha/beta versions) can be documented under the section Unreleased.

Possible types of changes are:

- `Added` for new features
- `Changed` for changes in existing functionality
- `Deprecated` for soon-to-be removed features
- `Removed` for now removed features
- `Fixed` for any bug fixes
- `Security` in case of vulnerabilities

9.1 1.3.0 - Unreleased

9.1.1 Added

- The `paragraph.wrap` virtual package. Any installed module can be imported under that package, resulting in all top-level callables being wrapped as paragraph ops.

9.1.2 Changed

- If the `__call__` method of an `Op` instance raises during execution of `paragraph.session.evaluate`, the latter catches the exception, raises a `RuntimeError` indicating the variable whose evaluation failed, and sets the original exception as the direct cause of the `RuntimeError`. Note that this currently only applies to single-threaded evaluations.

9.2 1.2.1 - 05.02.2020

9.2.1 Changed

- The `op` decorator now copies the `__doc__` attribute of the underlying function onto the wrapping `Op` instance

9.3 1.2.0 - 27.01.2020

9.3.1 Changed

- attribute `Op.thread_safe` is now a keyword-only argument of `__init__` to permit attributes without default values in derived classes.

9.4 1.1.1 - 23.01.2020

9.4.1 Fixed

- the API doc is now included

9.5 1.1.0 - 20.01.2020

9.5.1 Deprecated

- support for unresolved output variables in `session.evaluate`, use `sessions.solve` for that purpose
- support for arguments of type Variable in `Op.__call__`, use `Op.op` instead

9.5.2 Changed

- simplify `op` decorator
- simplify default implementation of `Op.__repr__`
- complete rewrite of the readme

9.5.3 Added

- method `Op.op` to be used in place of direct invocation of an Op instance when building a graph
- the `Variable` class, the `op` decorator and the functions `evaluate`, `apply`, `solve` and `solve_requirements` are now exposed at package level.

9.6 1.0.1 - 07.10.2020

9.6.1 Fixed

- mistyped variable name breaks `session.apply`

9.7 1.0.0 - 24.10.2019

9.7.1 Added

- initial version of source code

. Indices and tables ==========

- genindex
- modindex
- search

Python Module Index

p

Index