Documentation for pyadjoint

pyadjoint is a generic algorithmic differentiation framework. The aim of this page is to introduce the reader to how pyadjoint works and how it can be used. It can be useful to also have a look at the pyadjoint API. We stress that this is not an algorithmic differentiation tutorial. For an introduction to algorithmic differentiation see for example the book by Naumann

The core idea

In general, we will be interested in differentiating some output vector $y \in \mathbb{R}^m$ with respect to some input vector $x \in \mathbb{R}^n$. Assuming we can decompose the forward computations into

$$y = f(x) = g_k \circ g_{k-1} \circ \cdots g_1(x),$$

算子形成的有向无环图

then the derivative of the output in direction $\hat{x}$ is

$$\frac{\mathrm{d}y}{\mathrm{d}x}\hat{x} = \frac{\partial g_k(w_{k-1})}{\partial w_{k-1}} \frac{\partial g_{k-1}(w_{k-2})}{\partial w_{k-2}} \cdots \frac{\partial g_2(w_1)}{\partial w_1} \frac{\partial g_1(x)}{\partial x}\hat{x},$$

grad[f(g(x))]=grad[f(g(x))]+grad[g(x)]

for some intermediate solutions $w_i, i = 1, 2, \ldots, k-1$. In reverse mode algorithmic differentiation, we are interested in the adjoints:

$$\frac{\mathrm{d}y}{\mathrm{d}x}^* \bar{y} = \frac{\partial g_1(x)}{\partial x}^* \frac{\partial g_2(w_1)}{\partial w_1}^* \cdots \frac{\partial g_{k-1}(w_{k-2})}{\partial w_{k-2}}^* \frac{\partial g_k(w_{k-1})}{\partial w_{k-1}}^* \bar{y},$$

for some weights $\bar{y} \in \mathbb{R}^m$. In order to compute the adjoints we must somehow remember all the operations performed in the forward computations, and either remember or recompute the intermediate variables.

In pyadjoint, the `Tape` class is responsible for remembering these operations. The operations are stored in a `list`, and can be thought of as nodes in a directed acyclic graph. The `Tape` objects thus offer ways to do graph manipulations and visualisations.

The operations or nodes are represented by instances of subclasses of `Block`. These subclasses implement operation-specific methods, while the parent class, `Block`, implements common methods for use with the tape. These `Block` instances also keep track of their inputs and outputs, which are represented by `BlockVariable` instances.

For all this to be set up correctly, we require a way to populate the `Tape` and instantiate `Block` objects. To achieve this, we create *overloaded functions* that replace original functions and that are responsible for both of these tasks. An overloaded function should, for the user, act exactly as the original function.

In addition, to deal with mutating data objects used as inputs/outputs, all data types should have an overloaded counterpart. The overloaded data-types should inherit from both the original data-type and the pyadjoint class `OverloadedType`. `OverloadedType` ensures that the data-type has a `BlockVariable` instance attached, and declares some abstract methods for interaction with pyadjoint API functions that should be implemented by the specific data-types.

The core classes of pyadjoint are thus `Tape`, `Block`, `BlockVariable` and `OverloadedType`. We will now discuss each class individually, starting with `OverloadedType`.

OverloadedType

Block用于表示被记录的算子操作，它被tape记录以生成有向无环图，Block只保留input与output信息。OverloadedType用于重载操作符，它与Block是相互独立的。

The pyadjoint user-API provides several useful functions that act on the tape. For example, the function `taylor_test()` for verifying implementation, and `minimize()` in the optimization subpackage for minimizing functionals. To allow these functions to work without any knowledge of the structure of the data-types, some logic is moved into abstract methods of the `OverloadedType`, and are expected to be implemented for the individual data-types. At pyadjoint API you can see the individual abstract methods. Some methods are more important than others, because some of the abstract methods are only required for specific functions, while for instance `OverloadedType._ad_create_checkpoint()` and `OverloadedType._ad_restore_at_checkpoint()` are required for just working with the tape at all.

The `OverloadedType` class also has a single attribute, `block_variable`, which holds an instance of `BlockVariable`. In addition it defines the method `OverloadedType.create_block_variable()` which sets `block_variable` attribute to a

new `BlockVariable` instance, and returns it. This is used when adding the data-type as an output to a block. More information on that below.

To ensure that all pyadjoint specific methods are available, all data-type instances exposed to the end-user must be converted to overloaded versions. This is achieved through the `create_overloaded_object()` function, which combines a dictionary mapping original data-types to overloaded data-types, and the individually implemented `OverloadedType._ad_init_object()` method.

To populate the dictionary map, one must call `register_overloaded_type()`. This can conveniently be accomplished by using the function as a decorator when defining the overloaded data-type. In that case, you must use `OverloadedType` as the first base class, and the original data-type as second base class. Apart from implementing the abstract methods, one must also remember to call the constructor `OverloadedType.__init__()` in the overloaded data-type constructor.

BlockVariable

To track intermediate solutions, pyadjoint employs the class `BlockVariable`. Storing `interm_sol = y` does not guarantee that `interm_sol` remains the same until the end of the program execution if `y` is a mutable type. Thus, to ensure that the right values are kept, we create copies of the values used as input/output to operations.

Every time an instance of a data-type changes values, it should be assigned a new `BlockVariable`. Hence, `BlockVariable` can be thought of as an identifier for a specific version of a specific data-type object.

The `BlockVariable` class is quite simple. It holds a reference to the `OverloadedType` instance that created it, a checkpoint, some attributes for storing values in adjoint and tangent linear sweeps, and some flags.

The checkpoint is a copy of the values of the data-type (`OverloadedType`) instance. It does not have to be an exact copy of the instance. All that is required is that it is possible to restore an instance of the same type with the same values at a later point. This is implemented in the `OverloadedType._ad_create_checkpoint()` and `OverloadedType._ad_restore_at_checkpoint()` methods in the `OverloadedType` class. As an example, if a data-type was a function parameterised by a `float`, then a checkpont only requires storing this `float`, and the restoring method can create the same function using the same parameter value.

The attribute `tlm_value` holds the derivative direction for the forward mode AD sweep. This should be an instance of the same type as the corresponding `OverloadedType` instance.

The attribute `adj_value` holds the computed adjoint derivative action (can be thought of as the gradient). The type is in theory up to the block implementations, but to ensure compatibility across different blocks it should be an array-like type, such as a numpy array. Also it must be ensured that the choice of inner product is consistent between blocks. Thus, it is recommended that all blocks employ the l^2 inner product, i.e $(u, v)_{l^2} = u^T v$ where $u, v \in \mathbb{R}^n$. If the gradient with some other inner-product is desired, one can use Riesz representation theorem in the `OverloadedType._ad_convert_type()` method of `OverloadedType`.

The attribute `hessian_value` holds the computed hessian vector action. This should have the same type as `adj_value`.

Block

Before we go into how Blocks are implemented, let us take a look at a basic implementation of an overloaded function. Instead of using `overload_function()` we manually define the overloaded function in a similar way that the pyadjoint function would automatically do for you.

```
backend_example_function = example_function
def example_function(*args, **kwargs):
    annotate = annotate_tape(kwargs)
    if annotate:
        tape = get_working_tape()
        b_kwargs = ExampleBlock.pop_kwargs(kwargs)
        b_kwargs.update(kwargs)
        block = ExampleBlock(*args, **b_kwargs)
        tape.add_block(block)

    with stop_annotating():
        output = backend_example_function(*args, **kwargs)
    output = create_overloaded_object(output)

    if annotate:
        block.add_output(output.create_block_variable())
```

```
    return output
```

Let us go line by line through this. First we store a reference to the original function, then we start defining the overloaded function. Since overloaded functions can take some extra keyword arguments, one should use varying length keyword arguments in the function definition. Then we pass the keyword arguments to the pyadjoint function `annotate_tape()`. This will try to pop the keyword argument *annotate* from the keyword arguments dictionary, and return whether annotation is turned on. If annotation is turned on, we must add the operation to the tape. We first fetch the current tape using `get_working_tape()`, then we pop block-specific keyword arguments and merge them with the actual keyword arguments. These are then used when we instantiate the block, which in our example is `ExampleBlock`. Then the block instance is added to the tape.

No matter if we annotate or not, we must run the original function. To prevent the inner code of the original function to be annotated, we use the pyadjoint context manager `stop_annotating()`. After calling the original function, we convert the returned output to `OverloadedType`. Finally, if we are annotating then we create a new block variable for the output and add it as output of the block.

We now focus on the implementation of the block (`ExampleBlock` in the case above). The implementation of the constructor of the Block is largely up to the implementing user, as the main requirement is that the overloaded function and the block constructor are on the same page regarding how inputs/outputs are passed and what should be handled in the constructor and what is handled in the overloaded function.

For our example above, the constructor must first call the parent-class constructor, and also add the *dependencies* (inputs) using the `Block.add_dependency()` method. This method takes a block variable as input and appends it to a `list`, and thus it is important that all objects that are to be added to the dependencies should be an overloaded type. Below we show an example of a block constructor.

```
class ExampleBlock(Block):
    def __init__(self, *args, **kwargs):
    super(ExampleBlock, self).__init__()
    self.kwargs = kwargs
    for arg in args:
        self.add_dependency(arg.block_variable)
```

Note that not necessarily all arguments need to be dependencies. Only the inputs for which we wish to enable derivatives are strictly needed as dependencies.

Similarly to the dependencies, the output is also a list of block variables. Although it is often not needed, we can obtain the list of dependencies or outputs using the `Block.get_dependencies()` and `Block.get_outputs()` methods. It is important to note that the block only stores `BlockVariable` instances in these lists, and that to get the real values you need to access attributes of the `BlockVariable`. For example, to restore the checkpoint and get the restored object, use `x = block_variable.saved_output`.

The core methods of `Block` that allow for recomputations and derivatives to be computed are `Block.recompute()`, `Block.evaluate_adj()`, `Block.evaluate_tlm()` and `Block.evaluate_hessian()`. These methods are implemented in the abstract `Block` class, and by default delegate to the abstract methods `*_component()` (i.e `Block.evaluate_adj_component()`).

We first inspect how `Block.recompute()` works. The core idea is to use dependency checkpoints to compute new outputs and overwrite the output checkpoints with these new values. In the most basic form, the recompute method can be implemented as follows.

```
def recompute(self, markings=False):
    x = self.get_dependencies()[0].saved_output
    y = backend_example_function(x)
    self.get_outputs()[0].checkpoint = y
```

Here we have assumed that there is only one real dependency, hence `self.get_dependencies()` is a list of length one. Similarly we assume that this is the only input needed to the original function, and that the output is given explicitly through the return value of the original function. Lastly, we assume that the block has only one output and thus the length of `self.get_outputs()` is one.

The optional keyword argument `markings` is set to `True` when relevant block variables have been flagged. In that case, the recompute implementation can do optimizations by not recomputing outputs that are not relevant for what the user is interested in.

This unwrapping and working with attributes of `BlockVariable` instances may seem unnecessarily complicated, but it offers great flexibility. The `Block.recompute_component()` method tries to impose a more rigid structure, but can be

replaced by individual blocks by just overloading the `Block.recompute()` method directly.

The following is an example of the same implementation with `Block.recompute_component()`

```
def recompute_component(self, inputs, block_variable, idx, prepared):
    return backend_example_function(inputs[0])
```

Here the typically important variables are already sorted for you. `inputs` is a list of the new input values i.e the same as making a list of the `saved_output` of all the dependencies. Furthermore, each call to the `Block.recompute_compontent()` method is only for recomputing a single output, thus alleviating the need for code that optimizes based on block variable flags when `markings == True`. The `block_variable` parameter is the block variable of the output to recompute, while the `idx` is the index of the output in the `self.get_outputs()` list.

Sometimes you might want to do something once, that is common for all output recomputations. For example, your original function might return all the outputs, or you must prepare the input in a special way. Instead of doing this repeatedly for each call to `Block.recompute_component()`, one can implement the method `Block.prepare_recompute_component()`. This method by default returns `None`, but can return anything. The return value is supplied to the `prepared` argument of `Block.recompute_component()`. For each time `Block.recompute()` is called, `Block.prepare_recompute_component()` is called once and `Block.recompute_component()` is called once for each relevant output.

Now we take a look at `Block.evaluate_tlm()`. This method is used for the forward AD sweep and should compute the Jacobian vector product. More precisely, using the decomposition above, the method should compute

$$\hat{w}_{i+1} = \frac{\partial g_{i+1}(w_i)}{\partial w_i}\hat{w}_i$$

where $\hat{w}_i$ is some derivative direction, and g_{i+1} is the operation represented by the block. In `Block.evaluate_tlm()`, $\hat{w}_i$ has the same type as the function inputs (block dependencies) w_i. The following is a sketch of how `Block.evaluate_tlm()` can be implemented

```
def evaluate_tlm(self, markings=False):
    x = self.get_dependencies()[0].saved_output
    x_hat = self.get_dependencies()[0].tlm_value

    y_hat = derivative_example_function(x, x_hat)

    self.get_outputs()[0].add_tlm_output(y_hat)
```

We have again assumed that the example function only has one input and one output. Furthermore, we assume that we have implemented some derivative function in `derivative_example_function()`. The last line is the way to propagate the derivative directions forward in the tape. It essentially just adds the value to the `tlm_value` attribute of the output block variable, so that the next block can fetch it using `tlm_value`.

As with the recompute method, pyadjoint also offers a default `Block.evaluate_tlm()` implementation, that delegates to `Block.evaluate_tlm_component()` for each output. In our case, with only one output, the component method could look like this

```
def evaluate_tlm_component(self, inputs, tlm_inputs, block_variable, idx, prepared):
    return derivative_example_function(inputs[0], tlm_inputs[0])
```

The `prepared` parameter can be populated in the `Block.prepare_evaluate_tlm()` method.

`Block.evaluate_adj()` is responsible for computing the adjoint action or vector Jacobian product. Using the notation above, `Block.evaluate_adj()` should compute the following

$$\bar{w}_{i-1} = \frac{\partial g_i(w_{i-1})}{\partial w_{i-1}}^* \bar{w}_i$$

where the adjoint operator should be defined through the l^2 inner product. Assuming $g_i : \mathbb{R}^n \to \mathbb{R}^m$, then the adjoint should be defined by

$$(\frac{\partial g_i(w_{i-1})}{\partial w_{i-1}}u, v)_{\mathbb{R}^m} = (u, \frac{\partial g_i(w_{i-1})}{\partial w_{i-1}}^* v)_{\mathbb{R}^n}$$

for all $u \in \mathbb{R}^n, v \in \mathbb{R}^m$. Where $(a, b)_{\mathbb{R}^k} = a^T b$ for all $a, b \in \mathbb{R}^k, k \in \mathbb{N}$.

Using the same assumptions as earlier the implementation could look similar to this

```
def evaluate_adj(self, markings=False):
    y_bar = self.get_outputs()[0].adj_value
    x = self.get_dependencies()[0].saved_output

    x_bar = derivative_adj_example_function(x, y_bar)

    self.get_dependencies()[0].add_adj_output(x_bar)
```

There is also a default implementation for `Block.evaluate_adj()`, that calls the method `Block.evaluate_adj_component()` for each relevant dependency. This method could be implemented as follows

```
def evaluate_adj_component(self, inputs, adj_inputs, block_variable, idx, prepared):
    return derivative_adj_example_function(inputs[0], adj_inputs[0])
```

If there is any common computations across dependencies, these can be implemented in `Block.prepare_evaluate_adj()`.

Tape

As we have seen, we store the created block instances in a `Tape` instance. Each `Tape` instance holds a `list` of the block instances added to it. There can exists multiple `Tape` instances, but only one can be the current *working tape*. The working tape is the tape which is annotated to, i.e in which we will store any block instances created. It is also the tape that is by default interacted with when you run different pyadjoint functions that rely on a tape. The current working tape can be set and retrieved with the functions `set_working_tape()` and `get_working_tape()`.

Annotation can be temporarily disabled using `pause_annotation()` and enabled again using `continue_annotation()`. Note that if you call `pause_annotation()` twice, then `continue_annotation()` must be called twice to enable annotation. Due to this, the recommended annotation control functions are `stop_annotating` and `no_annotations()`. `stop_annotating` is a context manager and should be used as follows

```
with stop_annotating():
    # Code without annotation
    ...
```

`no_annotations()` is a decorator for disabling annotation within functions or methods. To check if annotation is enabled, use the function `annotate_tape()`.

Apart from storing the block instances, the `Tape` class offers a few methods for interaction with the computational graph. `Tape.visualise()` can be used to visualise the computational graph in a graph format. This can be useful for debugging purposes. `Tape.optimize()` offers a way to remove block instances that are not required for a reduced function. For optimizing the tape based on either a reduced output or input space, use the methods `Tape.optimize_for_functionals()` and `Tape.optimize_for_controls()`. Because these optimize methods mutate the tape, it can be useful to use the `Tape.copy()` method to keep a copy of the original list of block instances. To add block instances to the tape and retrieve the list of block instances, use `Tape.add_block()` and `Tape.get_blocks()`.

Other `Tape` methods are primarily used internally and users will rarely access these directly. However, it can be useful to know and use these methods when implementing custom overloaded functions. The tape instance methods that activate the `Block.evaluate_adj()` and `Block.evaluate_tlm()` methods are `Tape.evaluate_adj()`, `Tape.evaluate_tlm()`. These methods just iterate over all the blocks and call the corresponding evaluate method of the block. Usually some initialization is required, which is why these methods will likely not be called directly by the user. For example, for the backward sweep (`Tape.evaluate_adj()`) to work, you must initialize your functional adjoint value with the value 1. This is the default behaviour of the `compute_gradient()` function.

Similarly, to run the `Tape.evaluate_tlm()` properly, a direction, $\hat{x}$, must be specified. This can be done as follows

```
y = example_function(x)
x.block_variable.tlm_value = x_hat
tape = get_working_tape()
tape.evaluate_tlm()
dydx = y.block_variable.tlm_value
```

In a similar way, one can compute the gradient without using `compute_gradient()`

```
y = example_function(x)
y.block_variable.adj_value = y_bar
tape = get_working_tape()
tape.evaluate_adj()
grady = x.block_variable.adj_value
```

Where `y_bar` could be 1 if `y` is a float. However, `compute_gradient()` also performs other convenient operations. For example, it utilizes the markings flag in the `Block.evaluate_adj()` method. The markings are applied using the context manager `Tape.marked_nodes()`. In addition, `compute_gradient()` converts `adj_value` to overloaded types using the `OverloadedType._ad_convert_type()` method.