Source code for torch.autograd.variable

```
import sys
import torch
import torch._C as _C
from collections import OrderedDict
import torch.sparse as sparse
import torch.utils.hooks as hooks
import warnings
import weakref
from torch._six import imap
from torch._C import _add_docstr


class Variable(_C._VariableBase):                                                        [docs]
    """Wraps a tensor and records the operations applied to it.

    Variable is a thin wrapper around a Tensor object, that also holds
    the gradient w.r.t. to it, and a reference to a function that created it.
    This reference allows retracing the whole chain of operations that
    created the data. If the Variable has been created by the user, its grad_fn
    will be ``None`` and we call such objects *leaf* Variables.

    Since autograd only supports scalar valued function differentiation, grad
    size always matches the data size. Also, grad is normally only allocated
    for leaf variables, and will be always zero otherwise.

    Attributes:
        data: Wrapped tensor of any type.
        grad: Variable holding the gradient of type and location matching
            the ``.data``.  This attribute is lazily allocated and can't
            be reassigned.
        requires_grad: Boolean indicating whether the Variable has been
            created by a subgraph containing any Variable, that requires it.
            See :ref:`excluding-subgraphs` for more details.
            Can be changed only on leaf Variables.
        is_leaf: Boolean indicating if the Variable is a graph leaf (i.e
            if it was created by the user).
        grad_fn: Gradient function graph trace.

    Parameters:
        data (any tensor class): Tensor to wrap.
        requires_grad (bool): Value of the requires_grad flag. **Keyword only.**
    """

    def __deepcopy__(self, memo):
        if not self.is_leaf:
            raise RuntimeError("Only Variables created explicitly by the user "
                               "(graph leaves) support the deepcopy protocol at the moment")
        result = type(self)(self.data.clone())
        result.requires_grad = self.requires_grad
        memo[id(self)] = result
        return result

    def __reduce_ex__(self, proto):
        state = (self.requires_grad, False, self._backward_hooks)
        if proto > 1:
            return type(self), (self.data,), state
        if sys.version_info[0] == 2:
            from copy_reg import __newobj__
        else:
            from copyreg import __newobj__
        return __newobj__, (type(self), self.data), state

    def __setstate__(self, state):
        if len(state) == 5:
            # legacy serialization of Variable
            self.data = state[0]
```

```
            state = (state[3], state[4], state[2])
        if not self.is_leaf:
            raise RuntimeError('__setstate__ can be only called on leaf variables')
        self.requires_grad, _, self._backward_hooks = state

    def __repr__(self):
        return 'Variable containing:' + self.data.__repr__()

    def backward(self, gradient=None, retain_graph=None, create_graph=False):          [docs]
        """Computes the gradient of current variable w.r.t. graph leaves.

        The graph is differentiated using the chain rule. If the variable is
        non-scalar (i.e. its data has more than one element) and requires
        gradient, the function additionally requires specifying ``gradient``.
        It should be a tensor of matching type and location, that contains
        the gradient of the differentiated function w.r.t. ``self``.

        This function accumulates gradients in the leaves - you might need to
        zero them before calling it.

        Arguments:
            gradient (Tensor, Variable or None): Gradient w.r.t. the
                variable. If it is a tensor, it will be automatically converted
                to a Variable that does not require grad unless ``create_graph`` is True.
                None values can be specified for scalar Variables or ones that
                don't require grad. If a None value would be acceptable then
                this argument is optional.
            retain_graph (bool, optional): If ``False``, the graph used to compute
                the grads will be freed. Note that in nearly all cases setting
                this option to True is not needed and often can be worked around
                in a much more efficient way. Defaults to the value of
                ``create_graph``.
            create_graph (bool, optional): If ``True``, graph of the derivative will
                be constructed, allowing to compute higher order derivative
                products. Defaults to ``False``.
        """
        torch.autograd.backward(self, gradient, retain_graph, create_graph)


    def register_hook(self, hook):                                                     [docs]
        """Registers a backward hook.

        The hook will be called every time a gradient with respect to the
        variable is computed. The hook should have the following signature::

            hook(grad) -> Variable or None

        The hook should not modify its argument, but it can optionally return
        a new gradient which will be used in place of :attr:`grad`.

        This function returns a handle with a method ``handle.remove()``
        that removes the hook from the module.

        Example:
            >>> v = Variable(torch.Tensor([0, 0, 0]), requires_grad=True)
            >>> h = v.register_hook(lambda grad: grad * 2)  # double the gradient
            >>> v.backward(torch.Tensor([1, 1, 1]))
            >>> v.grad.data
             2
             2
             2
            [torch.FloatTensor of size 3]
            >>> h.remove()  # removes the hook
        """
        if not self.requires_grad:
            raise RuntimeError("cannot register a hook on a variable that "
                               "doesn't require gradient")
```

```
        if self._backward_hooks is None:
            self._backward_hooks = OrderedDict()
            if self.grad_fn is not None:
                self.grad_fn._register_hook_dict(self)
        handle = hooks.RemovableHandle(self._backward_hooks)
        self._backward_hooks[handle.id] = hook
        return handle


    def reinforce(self, reward):
        def trim(str):
            return '\n'.join([line.strip() for line in str.split('\n')])

        raise RuntimeError(trim(r"""reinforce() was removed.
            Use torch.distributions instead.
            See http://pytorch.org/docs/master/distributions.html

            Instead of:

            probs = policy_network(state)
            action = probs.multinomial()
            next_state, reward = env.step(action)
            action.reinforce(reward)
            action.backward()

            Use:

            probs = policy_network(state)
            # NOTE: categorical is equivalent to what used to be called multinomial
            m = torch.distributions.Categorical(probs)
            action = m.sample()
            next_state, reward = env.step(action)
            loss = -m.log_prob(action) * reward
            loss.backward()
        """))

    detach = _add_docstr(_C._VariableBase.detach, r"""
    Returns a new Variable, detached from the current graph.

    The result will never require gradient.

    .. note::

      Returned Variable uses the same data tensor, as the original one, and
      in-place modifications on either of them will be seen, and may trigger
      errors in correctness checks.
    """)

    detach_ = _add_docstr(_C._VariableBase.detach_, r"""
    Detaches the Variable from the graph that created it, making it a leaf.
    Views cannot be detached in-place.
    """)

    def retain_grad(self):                                                  [docs]
        """Enables .grad attribute for non-leaf Variables."""
        if self.grad_fn is None:  # no-op for leaves
            return
        if not self.requires_grad:
            raise RuntimeError("can't retain_grad on Variable that has requires_grad=False")
        if hasattr(self, 'retains_grad'):
            return
        weak_self = weakref.ref(self)

        def retain_grad_hook(grad):
            var = weak_self()
            if var is None:
                return
```

```
            if var._grad is None:
                var._grad = grad.clone()
            else:
                var._grad = var._grad + grad

        self.register_hook(retain_grad_hook)
        self.retains_grad = True


    def type_as(self, other):
        if torch.is_tensor(other):
            other = Variable(other)
        return super(Variable, self).type_as(other)

    def is_pinned(self):                                                    [docs]
        r"""Returns true if this tensor resides in pinned memory"""
        storage = self.storage()
        return storage.is_pinned() if storage else False


    def is_shared(self):                                                    [docs]
        r"""Checks if tensor is in shared memory.

        This is always ``True`` for CUDA tensors.
        """
        return self.storage().is_shared()


    def share_memory_(self):                                                [docs]
        r"""Moves the underlying storage to shared memory.

        This is a no-op if the underlying storage is already in shared memory
        and for CUDA tensors. Tensors in shared memory cannot be resized.
        """
        self.storage().share_memory_()


    def prod(self, dim=None, keepdim=None):
        return Prod.apply(self, dim, keepdim)

    def view_as(self, tensor):
        return self.view(tensor.size())

    def repeat(self, *repeats):
        if len(repeats) == 1 and isinstance(repeats[0], torch.Size):
            repeats = repeats[0]
        else:
            repeats = torch.Size(repeats)
        return Repeat.apply(self, repeats)

    def btrifact(self, info=None, pivot=True):
        if info is not None:
            warnings.warn("info option in btrifact is deprecated and will be removed in
v0.4, "
                          "consider using btrifact_with_info instead")
            factorization, pivots, _info = super(Variable,
self).btrifact_with_info(pivot=pivot)
            if not isinstance(info, Variable) or info.type() != _info.type():
                raise ValueError('btrifact expects info to be a Variable of IntTenor')
            info.data.copy_(_info.data)
            return factorization, pivots
        else:
            return super(Variable, self).btrifact(pivot=pivot)

    def cumprod(self, dim):
        return Cumprod.apply(self, dim)
```

```
    def resize(self, *sizes):
        return Resize.apply(self, sizes)

    def resize_as(self, variable):
        return Resize.apply(self, variable.size())

    def index_add(self, dim, index, tensor):
        return self.clone().index_add_(dim, index, tensor)

    def index_copy(self, dim, index, tensor):
        return self.clone().index_copy_(dim, index, tensor)

    def index_fill(self, dim, index, value):
        return self.clone().index_fill_(dim, index, value)

    def scatter(self, dim, index, source):
        return self.clone().scatter_(dim, index, source)

    def scatter_add(self, dim, index, source):
        return self.clone().scatter_add_(dim, index, source)

    def masked_copy(self, mask, variable):
        warnings.warn("masked_copy is deprecated and renamed to masked_scatter, and will be
removed in v0.3")
        return self.masked_scatter(mask, variable)

    def masked_copy_(self, mask, variable):
        warnings.warn("masked_copy_ is deprecated and renamed to masked_scatter_, and will
be removed in v0.3")
        return self.masked_scatter_(mask, variable)

    def masked_scatter(self, mask, variable):
        return self.clone().masked_scatter_(mask, variable)

    def masked_fill(self, mask, value):
        return self.clone().masked_fill_(mask, value)

    def expand_as(self, tensor):
        return self.expand(tensor.size())

    def __rsub__(self, other):
        return -self + other

    def __rdiv__(self, other):
        return self.reciprocal() * other
    __rtruediv__ = __rdiv__
    __itruediv__ = _C._VariableBase.__div__

    __pow__ = _C._VariableBase.pow

    def __ipow__(self, other):
        raise NotImplementedError("in-place pow not implemented")

    def __rpow__(self, other):
        return PowConstant.apply(other, self)

    __neg__ = _C._VariableBase.neg

    __eq__ = _C._VariableBase.eq
    __ne__ = _C._VariableBase.ne
    __lt__ = _C._VariableBase.lt
    __le__ = _C._VariableBase.le
    __gt__ = _C._VariableBase.gt
    __ge__ = _C._VariableBase.ge

    def __len__(self):
        return len(self.data)
```

```
    def __iter__(self):
        # NB: we use 'imap' and not 'map' here, so that in Python 2 we get a
        # generator and don't eagerly perform all the indexes.  This could
        # save us work, and also helps keep trace ordering deterministic
        # (e.g., if you zip(*hiddens), the eager map will force all the
        # indexes of hiddens[0] before hiddens[1], while the generator
        # map will interleave them.)
        return iter(imap(lambda i: self[i], range(self.size(0))))

    def __hash__(self):
        return id(self)

    def __dir__(self):
        variable_methods = dir(self.__class__)
        variable_methods.remove('volatile')  # deprecated
        attrs = list(self.__dict__.keys())
        keys = variable_methods + attrs
        return sorted(keys)

    # Numpy array interface, to support `numpy.asarray(tensor) -> ndarray`
    def __array__(self, dtype=None):
        if dtype is None:
            return self.cpu().numpy()
        else:
            return self.cpu().numpy().astype(dtype, copy=False)

    # Wrap Numpy array again in a suitable tensor when done, to support e.g.
    # `numpy.sin(tensor) -> tensor` or `numpy.greater(tensor, 0) -> ByteTensor`
    def __array_wrap__(self, array):
        if array.dtype == bool:
            # Workaround, torch has no built-in bool tensor
            array = array.astype('uint8')
        return Variable.from_numpy(array)

    class _torch(object):
        pass


for method in dir(Variable):
    # This will also wrap some methods that normally aren't part of the
    # functional interface, but we don't care, as they won't ever be used
    if method.startswith('_') or method.endswith('_'):
        continue
    if hasattr(Variable._torch, method):
        continue
    as_static = staticmethod(getattr(Variable, method))
    setattr(Variable._torch, method, as_static)


from ._functions import *
from torch._C import _ImperativeEngine as ImperativeEngine
Variable._execution_engine = ImperativeEngine()
```