import time

import logging

import warnings

import numpy as np

from .. import metric

from .. import ndarray

from ..context import cpu

from ..model import BatchEndParam

from ..initializer import Uniform

from ..io import DataDesc, DataIter, DataBatch

from ..base import \_as\_list

def \_check\_input\_names(symbol, names, typename, throw):

"""Check that all input names are in symbol's arguments."""

args = symbol.list\_arguments()

for name in names:

if name in args:

continue

candidates = [arg for arg in args if

not arg.endswith('\_weight') and

not arg.endswith('\_bias') and

not arg.endswith('\_gamma') and

not arg.endswith('\_beta')]

msg = "\033[91mYou created Module with Module(..., %s\_names=%s) but " \

"input with name '%s' is not found in symbol.list\_arguments(). " \

"Did you mean one of:\n\t%s\033[0m"%(

typename, str(names), name, '\n\t'.join(candidates))

if throw:

raise ValueError(msg)

else:

warnings.warn(msg)

def \_check\_names\_match(data\_names, data\_shapes, name, throw):

"""Check that input names matches input data descriptors."""

actual = [x[0] for x in data\_shapes]

if sorted(data\_names) != sorted(actual):

msg = "Data provided by %s\_shapes don't match names specified by %s\_names (%s vs. %s)"%(

name, name, str(data\_shapes), str(data\_names))

if throw:

raise ValueError(msg)

else:

warnings.warn(msg)

def \_parse\_data\_desc(data\_names, label\_names, data\_shapes, label\_shapes):

"""parse data\_attrs into DataDesc format and check that names match"""

data\_shapes = [x if isinstance(x, DataDesc) else DataDesc(\*x) for x in data\_shapes]

\_check\_names\_match(data\_names, data\_shapes, 'data', True)

if label\_shapes is not None:

label\_shapes = [x if isinstance(x, DataDesc) else DataDesc(\*x) for x in label\_shapes]

\_check\_names\_match(label\_names, label\_shapes, 'label', False)

else:

\_check\_names\_match(label\_names, [], 'label', False)

return data\_shapes, label\_shapes

class BaseModule(object):

"""The base class of a module.

A module represents a computation component. One can think of module as a computation machine.

A module can execute forward and backward passes and update parameters in a model.

We aim to make the APIs easy to use, especially in the case when we need to use the imperative

API to work with multiple modules (e.g. stochastic depth network).

A module has several states:

- Initial state: Memory is not allocated yet, so the module is not ready for computation yet.

- Binded: Shapes for inputs, outputs, and parameters are all known, memory has been allocated,

and the module is ready for computation.

- Parameters are initialized: For modules with parameters, doing computation before

initializing the parameters might result in undefined outputs.

- Optimizer is installed: An optimizer can be installed to a module. After this, the parameters

of the module can be updated according to the optimizer after gradients are computed

(forward-backward).

In order for a module to interact with others, it must be able to report the

following information in its initial state (before binding):

- `data\_names`: list of type string indicating the names of the required input data.

- `output\_names`: list of type string indicating the names of the required outputs.

After binding, a module should be able to report the following richer information:

- state information

- `binded`: `bool`, indicates whether the memory buffers needed for computation

have been allocated.

- `for\_training`: whether the module is bound for training.

- `params\_initialized`: `bool`, indicates whether the parameters of this module

have been initialized.

- `optimizer\_initialized`: `bool`, indicates whether an optimizer is defined

and initialized.

- `inputs\_need\_grad`: `bool`, indicates whether gradients with respect to the

input data are needed. Might be useful when implementing composition of modules.

- input/output information

- `data\_shapes`: a list of `(name, shape)`. In theory, since the memory is allocated,

we could directly provide the data arrays. But in the case of data parallelism,

the data arrays might not be of the same shape as viewed from the external world.

- `label\_shapes`: a list of `(name, shape)`. This might be `[]` if the module does

not need labels (e.g. it does not contains a loss function at the top), or a module

is not bound for training.

- `output\_shapes`: a list of `(name, shape)` for outputs of the module.

- parameters (for modules with parameters)

- `get\_params()`: return a tuple `(arg\_params, aux\_params)`. Each of those

is a dictionary of name to ``NDArray`` mapping. Those `NDArray` always lives on

CPU. The actual parameters used for computing might live on other devices (GPUs),

this function will retrieve (a copy of) the latest parameters.

- ``set\_params(arg\_params, aux\_params)``: assign parameters to the devices

doing the computation.

- ``init\_params(...)``: a more flexible interface to assign or initialize the parameters.

- setup

- `bind()`: prepare environment for computation.

- `init\_optimizer()`: install optimizer for parameter updating.

- `prepare()`: prepare the module based on the current data batch.

- computation

- `forward(data\_batch)`: forward operation.

- `backward(out\_grads=None)`: backward operation.

- `update()`: update parameters according to installed optimizer.

- `get\_outputs()`: get outputs of the previous forward operation.

- `get\_input\_grads()`: get the gradients with respect to the inputs computed

in the previous backward operation.

- `update\_metric(metric, labels, pre\_sliced=False)`: update performance metric

for the previous forward

computed results.

- other properties (mostly for backward compatibility)

- `symbol`: the underlying symbolic graph for this module (if any)

This property is not necessarily constant. For example, for `BucketingModule`,

this property is simply the \*current\* symbol being used. For other modules,

this value might not be well defined.

When those intermediate-level API are implemented properly, the following

high-level API will be automatically available for a module:

- `fit`: train the module parameters on a data set.

- `predict`: run prediction on a data set and collect outputs.

- `score`: run prediction on a data set and evaluate performance.

Examples

--------

>>> # An example of creating a mxnet module.

>>> import mxnet as mx

>>> data = mx.symbol.Variable('data')

>>> fc1 = mx.symbol.FullyConnected(data, name='fc1', num\_hidden=128)

>>> act1 = mx.symbol.Activation(fc1, name='relu1', act\_type="relu")

>>> fc2 = mx.symbol.FullyConnected(act1, name = 'fc2', num\_hidden = 64)

>>> act2 = mx.symbol.Activation(fc2, name='relu2', act\_type="relu")

>>> fc3 = mx.symbol.FullyConnected(act2, name='fc3', num\_hidden=10)

>>> out = mx.symbol.SoftmaxOutput(fc3, name = 'softmax')

>>> mod = mx.mod.Module(out)

"""

def \_\_init\_\_(self, logger=logging):

self.logger = logger

self.binded = False

self.for\_training = False

self.inputs\_need\_grad = False

self.params\_initialized = False

self.optimizer\_initialized = False

self.\_symbol = None

self.\_total\_exec\_bytes = 0

################################################################################

# High Level API

################################################################################

def forward\_backward(self, data\_batch):

"""A convenient function that calls both ``forward`` and ``backward``."""

self.forward(data\_batch, is\_train=True)

self.backward()

def score(self, eval\_data, eval\_metric, num\_batch=None, batch\_end\_callback=None,

score\_end\_callback=None,

reset=True, epoch=0, sparse\_row\_id\_fn=None):

"""Runs prediction on ``eval\_data`` and evaluates the performance according to

the given ``eval\_metric``.

Checkout `Module Tutorial <https://mxnet.apache.org/api/python/tutorials/packages/module/index.html>`\_

to see an end-to-end use-case.

Parameters

----------

eval\_data : DataIter

Evaluation data to run prediction on.

eval\_metric : EvalMetric or list of EvalMetrics

Evaluation metric to use.

num\_batch : int

Number of batches to run. Defaults to ``None``, indicating run until the `DataIter`

finishes.

batch\_end\_callback : function

Could also be a list of functions.

reset : bool

Defaults to ``True``. Indicates whether we should reset `eval\_data` before starting

evaluating.

epoch : int

Defaults to 0. For compatibility, this will be passed to callbacks (if any).

During training, this will correspond to the training epoch number.

sparse\_row\_id\_fn : A callback function

The function takes `data\_batch` as an input and returns a dict of

str -> NDArray. The resulting dict is used for pulling row\_sparse

parameters from the kvstore, where the str key is the name of the param,

and the value is the row id of the param to pull.

Examples

--------

>>> # An example of using score for prediction.

>>> # Evaluate accuracy on val\_dataiter

>>> metric = mx.metric.Accuracy()

>>> mod.score(val\_dataiter, metric)

>>> mod.score(val\_dataiter, ['mse', 'acc'])

"""

assert self.binded and self.params\_initialized

if reset:

eval\_data.reset()

if not isinstance(eval\_metric, metric.EvalMetric):

eval\_metric = metric.create(eval\_metric)

eval\_metric.reset()

actual\_num\_batch = 0

for nbatch, eval\_batch in enumerate(eval\_data):

if num\_batch is not None and nbatch == num\_batch:

break

self.prepare(eval\_batch, sparse\_row\_id\_fn=sparse\_row\_id\_fn)

self.forward(eval\_batch, is\_train=False)

if isinstance(eval\_batch, list):

self.update\_metric(eval\_metric, [eb.label for eb in eval\_batch], pre\_sliced=True)

else:

self.update\_metric(eval\_metric, eval\_batch.label)

if batch\_end\_callback is not None:

batch\_end\_params = BatchEndParam(epoch=epoch,

nbatch=nbatch,

eval\_metric=eval\_metric,

locals=locals())

for callback in \_as\_list(batch\_end\_callback):

callback(batch\_end\_params)

actual\_num\_batch += 1

if score\_end\_callback:

params = BatchEndParam(epoch=epoch,

nbatch=actual\_num\_batch,

eval\_metric=eval\_metric,

locals=locals())

for callback in \_as\_list(score\_end\_callback):

callback(params)

return eval\_metric.get\_name\_value()

def iter\_predict(self, eval\_data, num\_batch=None, reset=True, sparse\_row\_id\_fn=None):

"""Iterates over predictions.

Examples

--------

>>> for pred, i\_batch, batch in module.iter\_predict(eval\_data):

... # pred is a list of outputs from the module

... # i\_batch is a integer

... # batch is the data batch from the data iterator

Parameters

----------

eval\_data : DataIter

Evaluation data to run prediction on.

num\_batch : int

Default is ``None``, indicating running all the batches in the data iterator.

reset : bool

Default is ``True``, indicating whether we should reset the data iter before start

doing prediction.

sparse\_row\_id\_fn : A callback function

The function takes `data\_batch` as an input and returns a dict of

str -> NDArray. The resulting dict is used for pulling row\_sparse

parameters from the kvstore, where the str key is the name of the param,

and the value is the row id of the param to pull.

"""

assert self.binded and self.params\_initialized

if reset:

eval\_data.reset()

for nbatch, eval\_batch in enumerate(eval\_data):

if num\_batch is not None and nbatch == num\_batch:

break

self.prepare(eval\_batch, sparse\_row\_id\_fn=sparse\_row\_id\_fn)

self.forward(eval\_batch, is\_train=False)

pad = eval\_batch.pad

outputs = [out[0:out.shape[0]-pad] for out in self.get\_outputs()]

yield (outputs, nbatch, eval\_batch)

def predict(self, eval\_data, num\_batch=None, merge\_batches=True, reset=True,

always\_output\_list=False, sparse\_row\_id\_fn=None):

"""Runs prediction and collects the outputs.

When `merge\_batches` is ``True`` (by default), the return value will be a list

``[out1, out2, out3]``, where each element is formed by concatenating the outputs for

all the mini-batches. When `always\_output\_list` is ``False`` (as by default),

then in the case of a single output, `out1` is returned instead of ``[out1]``.

When `merge\_batches` is ``False``, the return value will be a nested list like

``[[out1\_batch1, out2\_batch1], [out1\_batch2], ...]``. This mode is useful because

in some cases (e.g. bucketing), the module does not necessarily produce the same

number of outputs.

The objects in the results have type `NDArray`. If you need to work with a numpy array,

just call ``.asnumpy()`` on each `NDArray`.

Parameters

----------

eval\_data : DataIter or NDArray or numpy array

Evaluation data to run prediction on.

num\_batch : int

Defaults to ``None``, indicates running all the batches in the data iterator.

merge\_batches : bool

Defaults to ``True``, see above for return values.

reset : bool

Defaults to ``True``, indicates whether we should reset the data iter before

doing prediction.

always\_output\_list : bool

Defaults to ``False``, see above for return values.

sparse\_row\_id\_fn : A callback function

The function takes `data\_batch` as an input and returns a dict of

str -> NDArray. The resulting dict is used for pulling row\_sparse

parameters from the kvstore, where the str key is the name of the param,

and the value is the row id of the param to pull.

Returns

-------

list of NDArray or list of list of NDArray

Prediction results.

Examples

--------

>>> # An example of using `predict` for prediction.

>>> # Predict on the first 10 batches of val\_dataiter

>>> mod.predict(eval\_data=val\_dataiter, num\_batch=10)

"""

assert self.binded and self.params\_initialized

if isinstance(eval\_data, (ndarray.NDArray, np.ndarray)):

if isinstance(eval\_data, np.ndarray):

eval\_data = ndarray.array(eval\_data)

self.forward(DataBatch([eval\_data]))

return self.get\_outputs()[0]

if not isinstance(eval\_data, DataIter):

raise ValueError('eval\_data must be of type NDArray or DataIter')

if reset:

eval\_data.reset()

output\_list = []

for nbatch, eval\_batch in enumerate(eval\_data):

if num\_batch is not None and nbatch == num\_batch:

break

self.prepare(eval\_batch, sparse\_row\_id\_fn=sparse\_row\_id\_fn)

self.forward(eval\_batch, is\_train=False)

pad = eval\_batch.pad

outputs = [out[0:out.shape[0]-pad].copy() for out in self.get\_outputs()]

output\_list.append(outputs)

if len(output\_list) == 0:

return output\_list

if merge\_batches:

num\_outputs = len(output\_list[0])

for out in output\_list:

assert len(out) == num\_outputs, \

'Cannot merge batches, as num of outputs is not the same ' + \

'in mini-batches. Maybe bucketing is used?'

output\_list2 = [ndarray.concatenate([out[i] for out in output\_list])

for i in range(num\_outputs)]

if num\_outputs == 1 and not always\_output\_list:

return output\_list2[0]

return output\_list2

return output\_list

def fit(self, train\_data, eval\_data=None, eval\_metric='acc',

epoch\_end\_callback=None, batch\_end\_callback=None, kvstore='local',

optimizer='sgd', optimizer\_params=(('learning\_rate', 0.01),),

eval\_end\_callback=None,

eval\_batch\_end\_callback=None, initializer=Uniform(0.01),

arg\_params=None, aux\_params=None, allow\_missing=False,

force\_rebind=False, force\_init=False, begin\_epoch=0, num\_epoch=None,

validation\_metric=None, monitor=None, sparse\_row\_id\_fn=None):

"""Trains the module parameters.

Checkout `Module Tutorial <https://mxnet.apache.org/api/python/tutorials/packages/module/index.html>`\_

to see an end-to-end use-case.

Parameters

----------

train\_data : DataIter

Train DataIter.

eval\_data : DataIter

If not ``None``, will be used as validation set and the performance

after each epoch will be evaluated.

eval\_metric : str or EvalMetric

Defaults to 'accuracy'. The performance measure used to display during training.

Other possible predefined metrics are:

'ce' (CrossEntropy), 'f1', 'mae', 'mse', 'rmse', 'top\_k\_accuracy'.

epoch\_end\_callback : function or list of functions

Each callback will be called with the current `epoch`, `symbol`, `arg\_params`

and `aux\_params`.

batch\_end\_callback : function or list of function

Each callback will be called with a `BatchEndParam`.

kvstore : str or KVStore

Defaults to 'local'.

optimizer : str or Optimizer

Defaults to 'sgd'.

optimizer\_params : dict

Defaults to ``(('learning\_rate', 0.01),)``. The parameters for

the optimizer constructor.

The default value is not a dict, just to avoid pylint warning on dangerous

default values.

eval\_end\_callback : function or list of function

These will be called at the end of each full evaluation, with the metrics over

the entire evaluation set.

eval\_batch\_end\_callback : function or list of function

These will be called at the end of each mini-batch during evaluation.

initializer : Initializer

The initializer is called to initialize the module parameters when they are

not already initialized.

arg\_params : dict

Defaults to ``None``, if not ``None``, should be existing parameters from a trained

model or loaded from a checkpoint (previously saved model). In this case,

the value here will be used to initialize the module parameters, unless they

are already initialized by the user via a call to `init\_params` or `fit`.

`arg\_params` has a higher priority than `initializer`.

aux\_params : dict

Defaults to ``None``. Similar to `arg\_params`, except for auxiliary states.

allow\_missing : bool

Defaults to ``False``. Indicates whether to allow missing parameters when `arg\_params`

and `aux\_params` are not ``None``. If this is ``True``, then the missing parameters

will be initialized via the `initializer`.

force\_rebind : bool

Defaults to ``False``. Whether to force rebinding the executors if already bound.

force\_init : bool

Defaults to ``False``. Indicates whether to force initialization even if the

parameters are already initialized.

begin\_epoch : int

Defaults to 0. Indicates the starting epoch. Usually, if resumed from a

checkpoint saved at a previous training phase at epoch N, then this value should be

N+1.

num\_epoch : int

Number of epochs for training.

sparse\_row\_id\_fn : A callback function

The function takes `data\_batch` as an input and returns a dict of

str -> NDArray. The resulting dict is used for pulling row\_sparse

parameters from the kvstore, where the str key is the name of the param,

and the value is the row id of the param to pull.

Examples

--------

>>> # An example of using fit for training.

>>> # Assume training dataIter and validation dataIter are ready

>>> # Assume loading a previously checkpointed model

>>> sym, arg\_params, aux\_params = mx.model.load\_checkpoint(model\_prefix, 3)

>>> mod.fit(train\_data=train\_dataiter, eval\_data=val\_dataiter, optimizer='sgd',

... optimizer\_params={'learning\_rate':0.01, 'momentum': 0.9},

... arg\_params=arg\_params, aux\_params=aux\_params,

... eval\_metric='acc', num\_epoch=10, begin\_epoch=3)

"""

assert num\_epoch is not None, 'please specify number of epochs'

self.bind(data\_shapes=train\_data.provide\_data, label\_shapes=train\_data.provide\_label,

for\_training=True, force\_rebind=force\_rebind)

if monitor is not None:

self.install\_monitor(monitor)

self.init\_params(initializer=initializer, arg\_params=arg\_params, aux\_params=aux\_params,

allow\_missing=allow\_missing, force\_init=force\_init)

self.init\_optimizer(kvstore=kvstore, optimizer=optimizer,

optimizer\_params=optimizer\_params)

if validation\_metric is None:

validation\_metric = eval\_metric

if not isinstance(eval\_metric, metric.EvalMetric):

eval\_metric = metric.create(eval\_metric)

################################################################################

# training loop

################################################################################

for epoch in range(begin\_epoch, num\_epoch):

tic = time.time()

eval\_metric.reset()

nbatch = 0

data\_iter = iter(train\_data)

end\_of\_batch = False

next\_data\_batch = next(data\_iter)

while not end\_of\_batch:

data\_batch = next\_data\_batch

if monitor is not None:

monitor.tic()

self.forward\_backward(data\_batch)

self.update()

if isinstance(data\_batch, list):

self.update\_metric(eval\_metric,

[db.label for db in data\_batch],

pre\_sliced=True)

else:

self.update\_metric(eval\_metric, data\_batch.label)

try:

# pre fetch next batch

next\_data\_batch = next(data\_iter)

self.prepare(next\_data\_batch, sparse\_row\_id\_fn=sparse\_row\_id\_fn)

except StopIteration:

end\_of\_batch = True

if monitor is not None:

monitor.toc\_print()

if end\_of\_batch:

eval\_name\_vals = eval\_metric.get\_global\_name\_value()

if batch\_end\_callback is not None:

batch\_end\_params = BatchEndParam(epoch=epoch, nbatch=nbatch,

eval\_metric=eval\_metric,

locals=locals())

for callback in \_as\_list(batch\_end\_callback):

callback(batch\_end\_params)

nbatch += 1

# one epoch of training is finished

for name, val in eval\_name\_vals:

self.logger.info('Epoch[%d] Train-%s=%f', epoch, name, val)

toc = time.time()

self.logger.info('Epoch[%d] Time cost=%.3f', epoch, (toc-tic))

# sync aux params across devices

arg\_params, aux\_params = self.get\_params()

self.set\_params(arg\_params, aux\_params)

if epoch\_end\_callback is not None:

for callback in \_as\_list(epoch\_end\_callback):

callback(epoch, self.symbol, arg\_params, aux\_params)

#----------------------------------------

# evaluation on validation set

if eval\_data:

res = self.score(eval\_data, validation\_metric,

score\_end\_callback=eval\_end\_callback,

batch\_end\_callback=eval\_batch\_end\_callback, epoch=epoch)

#TODO: pull this into default

for name, val in res:

self.logger.info('Epoch[%d] Validation-%s=%f', epoch, name, val)

# end of 1 epoch, reset the data-iter for another epoch

train\_data.reset()

################################################################################

# Symbol information

################################################################################

@property

def data\_names(self):

"""A list of names for data required by this module."""

raise NotImplementedError()

@property

def output\_names(self):

"""A list of names for the outputs of this module."""

raise NotImplementedError()

################################################################################

# Input/Output information

################################################################################

@property

def data\_shapes(self):

"""A list of (name, shape) pairs specifying the data inputs to this module."""

raise NotImplementedError()

@property

def label\_shapes(self):

"""A list of (name, shape) pairs specifying the label inputs to this module.

If this module does not accept labels -- either it is a module without loss

function, or it is not bound for training, then this should return an empty

list ``[]``.

"""

raise NotImplementedError()

@property

def output\_shapes(self):

"""A list of (name, shape) pairs specifying the outputs of this module."""

raise NotImplementedError()

################################################################################

# Parameters of a module

################################################################################

def get\_params(self):

"""Gets parameters, those are potentially copies of the actual parameters used

to do computation on the device.

Returns

-------

``(arg\_params, aux\_params)``

A pair of dictionaries each mapping parameter names to NDArray values.

Examples

--------

>>> # An example of getting module parameters.

>>> print mod.get\_params()

({'fc2\_weight': <NDArray 64x128 @cpu(0)>, 'fc1\_weight': <NDArray 128x100 @cpu(0)>,

'fc3\_bias': <NDArray 10 @cpu(0)>, 'fc3\_weight': <NDArray 10x64 @cpu(0)>,

'fc2\_bias': <NDArray 64 @cpu(0)>, 'fc1\_bias': <NDArray 128 @cpu(0)>}, {})

"""

raise NotImplementedError()

def init\_params(self, initializer=Uniform(0.01), arg\_params=None, aux\_params=None,

allow\_missing=False, force\_init=False, allow\_extra=False):

"""Initializes the parameters and auxiliary states.

Parameters

----------

initializer : Initializer

Called to initialize parameters if needed.

arg\_params : dict

If not ``None``, should be a dictionary of existing `arg\_params`. Initialization

will be copied from that.

aux\_params : dict

If not ``None``, should be a dictionary of existing `aux\_params`. Initialization

will be copied from that.

allow\_missing : bool

If ``True``, params could contain missing values, and the initializer will be

called to fill those missing params.

force\_init : bool

If ``True``, `force\_init` will force re-initialize even if already initialized.

allow\_extra : boolean, optional

Whether allow extra parameters that are not needed by symbol.

If this is True, no error will be thrown when arg\_params or aux\_params

contain extra parameters that is not needed by the executor.

Examples

--------

>>> # An example of initializing module parameters.

>>> mod.init\_params()

"""

raise NotImplementedError()

def set\_params(self, arg\_params, aux\_params, allow\_missing=False, force\_init=True,

allow\_extra=False):

"""Assigns parameter and aux state values.

Parameters

----------

arg\_params : dict

Dictionary of name to value (`NDArray`) mapping.

aux\_params : dict

Dictionary of name to value (`NDArray`) mapping.

allow\_missing : bool

If ``True``, params could contain missing values, and the initializer will be

called to fill those missing params.

force\_init : bool

If ``True``, will force re-initialize even if already initialized.

allow\_extra : boolean, optional

Whether allow extra parameters that are not needed by symbol.

If this is True, no error will be thrown when arg\_params or aux\_params

contain extra parameters that is not needed by the executor.

Examples

--------

>>> # An example of setting module parameters.

>>> sym, arg\_params, aux\_params = mx.model.load\_checkpoint(model\_prefix, n\_epoch\_load)

>>> mod.set\_params(arg\_params=arg\_params, aux\_params=aux\_params)

"""

self.init\_params(initializer=None, arg\_params=arg\_params, aux\_params=aux\_params,

allow\_missing=allow\_missing, force\_init=force\_init,

allow\_extra=allow\_extra)

def save\_params(self, fname):

"""Saves model parameters to file.

Parameters

----------

fname : str

Path to output param file.

Examples

--------

>>> # An example of saving module parameters.

>>> mod.save\_params('myfile')

"""

arg\_params, aux\_params = self.get\_params()

save\_dict = {('arg:%s' % k) : v.as\_in\_context(cpu()) for k, v in arg\_params.items()}

save\_dict.update({('aux:%s' % k) : v.as\_in\_context(cpu()) for k, v in aux\_params.items()})

ndarray.save(fname, save\_dict)

def load\_params(self, fname):

"""Loads model parameters from file.

Parameters

----------

fname : str

Path to input param file.

Examples

--------

>>> # An example of loading module parameters.

>>> mod.load\_params('myfile')

"""

save\_dict = ndarray.load(fname)

arg\_params = {}

aux\_params = {}

for k, value in save\_dict.items():

arg\_type, name = k.split(':', 1)

if arg\_type == 'arg':

arg\_params[name] = value

elif arg\_type == 'aux':

aux\_params[name] = value

else:

raise ValueError("Invalid param file " + fname)

self.set\_params(arg\_params, aux\_params)

def get\_states(self, merge\_multi\_context=True):

"""Gets states from all devices

If `merge\_multi\_context` is ``True``, returns output of form ``[out1, out2]``.

Otherwise, it returns output of the form

``[[out1\_dev1, out1\_dev2], [out2\_dev1, out2\_dev2]]``.

All output elements are `NDArray`.

Parameters

----------

merge\_multi\_context : bool

Defaults to ``True``. In the case when data-parallelism is used, the states

will be collected from multiple devices. A ``True`` value indicates that we

should merge the collected results so that they look like from a single

executor.

Returns

-------

A list of ``NDArray`` or a list of list of ``NDArray``.

"""

assert self.binded and self.params\_initialized

assert not merge\_multi\_context

return []

def set\_states(self, states=None, value=None):

"""Sets value for states. Only one of states & value can be specified.

Parameters

----------

states : list of list of NDArray

Source states arrays formatted like

``[[state1\_dev1, state1\_dev2], [state2\_dev1, state2\_dev2]]``.

value : number

A single scalar value for all state arrays.

"""

assert self.binded and self.params\_initialized

assert not states and not value

def install\_monitor(self, mon):

"""Installs monitor on all executors."""

raise NotImplementedError()

################################################################################

# Computations

################################################################################

# pylint: disable=unused-argument

def prepare(self, data\_batch, sparse\_row\_id\_fn=None):

'''Prepares the module for processing a data batch.

Usually involves switching bucket and reshaping.

For modules that contain `row\_sparse` parameters in KVStore,

it prepares the `row\_sparse` parameters based on the sparse\_row\_id\_fn.

When KVStore is used to update parameters for multi-device or multi-machine training,

a copy of the parameters are stored in KVStore. Note that for `row\_sparse` parameters,

the `update()` updates the copy of parameters in KVStore, but doesn't broadcast

the updated parameters to all devices / machines. The `prepare` function is used to

broadcast `row\_sparse` parameters with the next batch of data.

Parameters

----------

data\_batch : DataBatch

The current batch of data for forward computation.

sparse\_row\_id\_fn : A callback function

The function takes `data\_batch` as an input and returns a dict of

str -> NDArray. The resulting dict is used for pulling row\_sparse

parameters from the kvstore, where the str key is the name of the param,

and the value is the row id of the param to pull.

'''

if sparse\_row\_id\_fn is not None:

warnings.warn(UserWarning("sparse\_row\_id\_fn is not invoked for BaseModule."))

# pylint: enable=unused-argument

def forward(self, data\_batch, is\_train=None):

"""Forward computation. It supports data batches with different shapes, such as

different batch sizes or different image sizes.

If reshaping of data batch relates to modification of symbol or module, such as

changing image layout ordering or switching from training to predicting, module

rebinding is required.

Parameters

----------

data\_batch : DataBatch

Could be anything with similar API implemented.

is\_train : bool

Default is ``None``, which means `is\_train` takes the value of ``self.for\_training``.

Examples

--------

>>> import mxnet as mx

>>> from collections import namedtuple

>>> Batch = namedtuple('Batch', ['data'])

>>> data = mx.sym.Variable('data')

>>> out = data \* 2

>>> mod = mx.mod.Module(symbol=out, label\_names=None)

>>> mod.bind(data\_shapes=[('data', (1, 10))])

>>> mod.init\_params()

>>> data1 = [mx.nd.ones((1, 10))]

>>> mod.forward(Batch(data1))

>>> print mod.get\_outputs()[0].asnumpy()

[[ 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]]

>>> # Forward with data batch of different shape

>>> data2 = [mx.nd.ones((3, 5))]

>>> mod.forward(Batch(data2))

>>> print mod.get\_outputs()[0].asnumpy()

[[ 2. 2. 2. 2. 2.]

[ 2. 2. 2. 2. 2.]

[ 2. 2. 2. 2. 2.]]

"""

raise NotImplementedError()

def backward(self, out\_grads=None):

"""Backward computation.

Parameters

----------

out\_grads : NDArray or list of NDArray, optional

Gradient on the outputs to be propagated back.

This parameter is only needed when bind is called

on outputs that are not a loss function.

Examples

--------

>>> # An example of backward computation.

>>> mod.backward()

>>> print mod.get\_input\_grads()[0].asnumpy()

[[[ 1.10182791e-05 5.12257748e-06 4.01927764e-06 8.32566820e-06

-1.59775993e-06 7.24269375e-06 7.28067835e-06 -1.65902311e-05

5.46342608e-06 8.44196393e-07]

...]]

"""

raise NotImplementedError()

def get\_outputs(self, merge\_multi\_context=True):

"""Gets outputs of the previous forward computation.

If `merge\_multi\_context` is ``True``, it is like ``[out1, out2]``. Otherwise,

it returns out put of form ``[[out1\_dev1, out1\_dev2], [out2\_dev1, out2\_dev2]]``.

All the output elements have type `NDArray`. When `merge\_multi\_context` is ``False``,

those `NDArray` instances might live on different devices.

Parameters

----------

merge\_multi\_context : bool

Defaults to ``True``. In the case when data-parallelism is used, the outputs

will be collected from multiple devices. A ``True`` value indicates that we

should merge the collected results so that they look like from a single

executor.

Returns

-------

list of `NDArray` or list of list of `NDArray`.

Output

Examples

--------

>>> # An example of getting forward output.

>>> print mod.get\_outputs()[0].asnumpy()

[[ 0.09999977 0.10000153 0.10000716 0.10000195 0.09999853 0.09999743

0.10000272 0.10000113 0.09999088 0.09999888]]

"""

raise NotImplementedError()

def get\_input\_grads(self, merge\_multi\_context=True):

"""Gets the gradients to the inputs, computed in the previous backward computation.

If `merge\_multi\_context` is ``True``, it is like ``[grad1, grad2]``. Otherwise, it

is like ``[[grad1\_dev1, grad1\_dev2], [grad2\_dev1, grad2\_dev2]]``. All the output

elements have type `NDArray`. When `merge\_multi\_context` is ``False``, those `NDArray`

instances might live on different devices.

Parameters

----------

merge\_multi\_context : bool

Defaults to ``True``. In the case when data-parallelism is used, the gradients

will be collected from multiple devices. A ``True`` value indicates that we

should merge the collected results so that they look like from a single

executor.

Returns

-------

list of NDArray or list of list of NDArray

Input gradients.

Examples

--------

>>> # An example of getting input gradients.

>>> print mod.get\_input\_grads()[0].asnumpy()

[[[ 1.10182791e-05 5.12257748e-06 4.01927764e-06 8.32566820e-06

-1.59775993e-06 7.24269375e-06 7.28067835e-06 -1.65902311e-05

5.46342608e-06 8.44196393e-07]

...]]

"""

raise NotImplementedError()

def update(self):

"""Updates parameters according to the installed optimizer and the gradients computed

in the previous forward-backward batch.

When KVStore is used to update parameters for multi-device or multi-machine training,

a copy of the parameters are stored in KVStore. Note that for `row\_sparse` parameters,

this function does update the copy of parameters in KVStore, but doesn't broadcast the

updated parameters to all devices / machines. Please call `prepare` to broadcast

`row\_sparse` parameters with the next batch of data.

Examples

--------

>>> # An example of updating module parameters.

>>> mod.init\_optimizer(kvstore='local', optimizer='sgd',

... optimizer\_params=(('learning\_rate', 0.01), ))

>>> mod.backward()

>>> mod.update()

>>> print mod.get\_params()[0]['fc3\_weight'].asnumpy()

[[ 5.86930104e-03 5.28078526e-03 -8.88729654e-03 -1.08308345e-03

6.13054074e-03 4.27560415e-03 1.53817423e-03 4.62131854e-03

4.69872449e-03 -2.42400169e-03 9.94111411e-04 1.12386420e-03

...]]

"""

raise NotImplementedError()

def update\_metric(self, eval\_metric, labels, pre\_sliced=False):

"""Evaluates and accumulates evaluation metric on outputs of the last forward

computation.

Parameters

----------

eval\_metric : EvalMetric

Evaluation metric to use.

labels : list of NDArray if `pre\_sliced` parameter is set to `False`,

list of lists of NDArray otherwise. Typically `data\_batch.label`.

pre\_sliced: bool

Whether the labels are already sliced per device (default: False).

Examples

--------

>>> # An example of updating evaluation metric.

>>> mod.forward(data\_batch)

>>> mod.update\_metric(metric, data\_batch.label)

"""

raise NotImplementedError()

################################################################################

# module setup

################################################################################

def bind(self, data\_shapes, label\_shapes=None, for\_training=True,

inputs\_need\_grad=False, force\_rebind=False, shared\_module=None,

grad\_req='write'):

"""Binds the symbols to construct executors. This is necessary before one

can perform computation with the module.

Parameters

----------

data\_shapes : list of (str, tuple) or DataDesc objects

Typically is ``data\_iter.provide\_data``. Can also be a list of

(data name, data shape).

label\_shapes : list of (str, tuple) or DataDesc objects

Typically is ``data\_iter.provide\_label``. Can also be a list of

(label name, label shape).

for\_training : bool

Default is ``True``. Whether the executors should be bind for training.

inputs\_need\_grad : bool

Default is ``False``. Whether the gradients to the input data need to be computed.

Typically this is not needed. But this might be needed when implementing composition

of modules.

force\_rebind : bool

Default is ``False``. This function does nothing if the executors are already

bound. But with this ``True``, the executors will be forced to rebind.

shared\_module : Module

Default is ``None``. This is used in bucketing. When not ``None``, the shared module

essentially corresponds to a different bucket -- a module with different symbol

but with the same sets of parameters (e.g. unrolled RNNs with different lengths).

grad\_req : str, list of str, dict of str to str

Requirement for gradient accumulation. Can be 'write', 'add', or 'null'

(default to 'write').

Can be specified globally (str) or for each argument (list, dict).

Examples

--------

>>> # An example of binding symbols.

>>> mod.bind(data\_shapes=[('data', (1, 10, 10))])

>>> # Assume train\_iter is already created.

>>> mod.bind(data\_shapes=train\_iter.provide\_data, label\_shapes=train\_iter.provide\_label)

"""

raise NotImplementedError()

def init\_optimizer(self, kvstore='local', optimizer='sgd',

optimizer\_params=(('learning\_rate', 0.01),), force\_init=False):

"""Installs and initializes optimizers, as well as initialize kvstore for

distributed training

Parameters

----------

kvstore : str or KVStore

Defaults to `'local'`.

optimizer : str or Optimizer

Defaults to `'sgd'`.

optimizer\_params : dict

Defaults to ``(('learning\_rate', 0.01),)``. The default value is not a dictionary,

just to avoid pylint warning of dangerous default values.

force\_init : bool

Defaults to ``False``, indicates whether to force re-initializing an optimizer

if it is already installed.

Examples

--------

>>> # An example of initializing optimizer.

>>> mod.init\_optimizer(optimizer='sgd', optimizer\_params=(('learning\_rate', 0.005),))

"""

raise NotImplementedError()

################################################################################

# misc

################################################################################

@property

def symbol(self):

"""Gets the symbol associated with this module.

Except for `Module`, for other types of modules (e.g. `BucketingModule`), this

property might not be a constant throughout its life time. Some modules might

not even be associated with any symbols.

"""

return self.\_symbol