# Source code for cntk.learners

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license. See LICENSE.md file in the project root
# for full license information.
# ==============================================================================

'''
A learner tunes a set of parameters during the training process. One can use
different learners for different sets of parameters. Currently, CNTK supports
the following learning algorithms:

- :func:`AdaDelta <adadelta>`
- :func:`AdaGrad <adagrad>`
- :func:`FSAdaGrad <fsadagrad>`
- :func:`Adam <adam>`
- :func:`MomentumSGD <momentum_sgd>`
- :func:`Nesterov <nesterov>`
- :func:`RMSProp <rmsprop>`
- :func:`SGD <sgd>`
- :func:`Learner with a customized update function <universal>`
'''


from enum import Enum, unique
import warnings
import numpy as np
import cntk.internal.utils as utils

from .. import cntk_py, NDArrayView, asarray
from cntk.internal import typemap
from ..internal.swig_helper import map_if_possible


@unique
class UnitType(Enum):                                                    [docs]

    '''
    Indicates whether the values in the schedule are specified on the per-sample or
    per-minibatch basis.
    '''

    sample = 'sample'
    '''
    Schedule contains per-sample values.
    '''

    minibatch = 'minibatch'
    '''
    Schedule contains per-minibatch values (and need to be re-scaled by the learner
    using the actual minibatch size in samples).
    '''






def default_unit_gain_value():                                           [docs]
    '''
    Returns true if by default momentum is applied in the unit-gain fashion.
    '''
    return cntk_py.default_unit_gain_value()



def set_default_unit_gain_value(value):                                  [docs]
    '''
    Sets globally default unit-gain flag value.
    '''
    cntk_py.set_default_unit_gain_value(value)


def default_use_mean_gradient_value():                                   [docs]
    '''
    Returns true if by default input gradient to learner is averaged.
    '''
    return cntk_py.default_use_mean_gradient_value()
```

```
def set_default_use_mean_gradient_value(value):                                              [docs]
    '''
    Sets globally default use_mean_gradient_value.
    '''
    cntk_py.set_default_use_mean_gradient_value(value)


# an internal method to verify that the learning rate schedule
# has a proper (per-sample or per-MB schedule) type and raise
# an exception otherwise


def _verify_learning_rate_type(learning_rate):
    if not isinstance(learning_rate,
                      (cntk_py.training_parameter_per_sample_schedule,
                       cntk_py.training_parameter_per_minibatch_schedule)):

        raise ValueError('learning_rate type (%s) not supported. '
                         'learning_rate must be a training schedule '
                         '(output of learning_rate_schedule() function)'
                         % type(learning_rate))

# an internal method to verify that the mometum schedule
# has a proper (per-MB or time-constant schedule) type and raise
# an exception otherwise


def _verify_momentum_type(momentum):
    if not isinstance(momentum,
                      (cntk_py.training_parameter_per_minibatch_schedule,
                       cntk_py.momentum_as_time_constant_schedule)):

        raise ValueError('momentum type (%s) not supported. '
                         'momentum must be a training schedule '
                         '(output of momentum_schedule() or '
                         'momentum_as_time_constant_schedule() function)'
                         % type(momentum))


class Learner(cntk_py.Learner):                                                              [docs]

    '''
    Abstraction for learning a subset of parameters of a learnable function using first order
gradient values.
    For example momentum, AdaGrad, RMSProp, etc. are different types of learners with their own
algorithms for
    learning parameter values using first order gradients.
    To instantiate a concrete learner, use the factory methods in this module.
    '''

    def update(self, gradient_values, training_sample_count):                               [docs]
        '''
        Update the parameters associated with this learner.

        Args:
            gradient_values (dict): maps :class:`~cntk.variables.Parameter` to
             a NumPy array containing the first order gradient values for the
             Parameter w.r.t. the training objective.
            training_sample_count (int): number of samples in the minibatch

        Returns:
            bool: `False` to indicate that learning has stopped for all of the parameters
associated with this learner
        '''
        var_nd_map = {var: NDArrayView.from_data(val) for var, val in
                      gradient_values.items()}

        return super(Learner, self)._update(var_nd_map, training_sample_count)


    @property
    @typemap
    def parameters(self):
        '''
        The set of parameters associated with this learner.
        '''
        return super(Learner, self).parameters()

    def reset_learning_rate(self, learning_rate):                                  [docs]
        '''
        Resets the learning rate. The new schedule is adjusted to be relative
        to the current number of elapsed samples/sweeps: the 0 offset in
        the new schedule corresponds to the current value of elapsed samples/sweeps,
        and it takes effect from the current position in the training process onwards.

        Args:
            learning_rate (output of :func:`learning_rate_schedule`)
             learning rate to reset to
        '''
        _verify_learning_rate_type(learning_rate)
        return super(Learner, self).reset_learning_rate(learning_rate)


    def learning_rate(self):                                                       [docs]
        '''
        Current learning rate schedule.
        '''
        return super(Learner, self).learning_rate()



class UserLearner(cntk_py.Learner):                                                [docs]

    '''
    Base class of all user-defined learners. To implement your own learning
    algorithm, derive from this class and override the :meth:`update`.

    Certain optimizers (such as AdaGrad) require additional storage.
    This can be allocated and initialized during construction.
    '''

    def __init__(self, parameters, lr_schedule, as_numpy=True):
        super(UserLearner, self).__init__(parameters, lr_schedule)
        self.as_numpy = as_numpy
        self.__disown__()

    def _update(self, gradient_values, training_sample_count, sweep_end):
        '''
        Update the parameters and related state associated with this learner.

        Args:
            gradient_values (dict): maps :class:`~cntk.variables.Parameter`
             to a NumPy array containing the gradient for the Parameter w.r.t.
             the training objective.
            training_sample_count (int): number of samples in the minibatch
            sweep_end (bool): if the data is fed by a conforming reader, this
             indicates whether a full pass over the dataset has just occurred.

        Returns:
            bool: `False` to indicate that learning has stopped for all of the
            parameters associated with this learner
        '''
        map_if_possible(gradient_values)

        if self.as_numpy:
            var_nd_map = {var: asarray(gradient_values[var]) \
                          for var, val in gradient_values.items()}
        else:
            var_nd_map = gradient_values

        return self.update(gradient_values, training_sample_count, sweep_end)

    def update(self, gradient_values, training_sample_count, sweep_end):          [docs]
        '''
        Update the parameters associated with this learner.

        Args:
            gradient_values (dict): maps :class:`~cntk.variables.Parameter` to
             a NumPy array containing the first order gradient values for the
             Parameter w.r.t. the training objective.
            training_sample_count (int): number of samples in the minibatch
            sweep_end (bool): if the data is fed by a conforming reader, this indicates
             whether a full pass over the dataset has just occurred.

        Returns:
            bool: `False` to indicate that learning has stopped for all of the
            parameters associated with this learner
        '''
        raise NotImplementedError('UserLearner.update must be overriden')



@typemap
def training_parameter_schedule(schedule, unit, epoch_size=None):                    [docs]
    '''
    Create a training parameter schedule containing either per-sample (default)
    or per-minibatch values.

    Examples:
        >>> # Use a fixed value 0.01 for all samples
        >>> s = training_parameter_schedule(0.01, UnitType.sample)
        >>> s[0], s[1]
        (0.01, 0.01)

        >>> # Use 0.01 for the first 1000 samples, then 0.001 for the remaining ones
        >>> s = training_parameter_schedule([0.01, 0.001], UnitType.sample, 1000)
        >>> s[0], s[1], s[1000], s[1001]
        (0.01, 0.01, 0.001, 0.001)

        >>> # Use 0.1 for the first 12 epochs, then 0.01 for the next 15,
        >>> # followed by 0.001 for the remaining ones, with a 100 samples in an epoch
        >>> s = training_parameter_schedule([(12, 0.1), (15, 0.01), (1, 0.001)],
UnitType.sample, 100)
        >>> s[0], s[1199], s[1200], s[2699], s[2700], s[5000]
        (0.1, 0.1, 0.01, 0.01, 0.001, 0.001)

    Args:
        schedule (float or list): if float, is the parameter schedule to be used
         for all samples. In case of list, the elements are used as the
         values for ``epoch_size`` samples. If list contains pair, the second element is
         used as a value for (``epoch_size`` x first element) samples
        unit (:class:`UnitType`): one of two
          * ``sample``: the returned schedule contains per-sample values
          * ``minibatch``: the returned schedule contains per-minibatch values.
        epoch_size (optional, int): number of samples as a scheduling unit.
         Parameters in the schedule change their values every ``epoch_size``
         samples. If no ``epoch_size`` is provided, this parameter is substituted
         by the size of the full data sweep, in which case the scheduling unit is
         the entire data sweep (as indicated by the MinibatchSource) and parameters
         change their values on the sweep-by-sweep basis specified by the
         ``schedule``.

    Returns:
        training parameter schedule

    See also:
        :func:`learning_rate_schedule`
    '''
    if unit == UnitType.sample:
        if isinstance(schedule, cntk_py.training_parameter_per_sample_schedule):
            return schedule
    else:
        if isinstance(schedule, cntk_py.training_parameter_per_minibatch_schedule):
            return schedule

    if isinstance(schedule, (int, float)):
        if epoch_size is not None:
            warnings.warn('When providing the schedule as a number, epoch_size is ignored',
RuntimeWarning)
        if UnitType(unit) is UnitType.sample:
            return cntk_py.training_parameter_per_sample_schedule(schedule)
        else:
            return cntk_py.training_parameter_per_minibatch_schedule(schedule)

    args = [schedule] if epoch_size is None else [schedule, epoch_size]

    if isinstance(schedule, list):
        if UnitType(unit) is UnitType.sample:
            return cntk_py.training_parameter_per_sample_schedule(*args)
        else:
            return cntk_py.training_parameter_per_minibatch_schedule(*args)

    raise ValueError(
        'schedule must be either a float or a list, not %s' % type(schedule))



@typemap
def learning_rate_schedule(lr, unit, epoch_size=None):                                       [docs]
    '''
    Create a learning rate schedule (using the same semantics as
    :func:`training_parameter_schedule`).

    Args:
        lr (float or list): see parameter ``schedule`` in
         :func:`training_parameter_schedule`.
        unit (:class:`UnitType`): see parameter
         ``unit`` in :func:`training_parameter_schedule`.
        epoch_size (int): see parameter ``epoch_size`` in
         :func:`training_parameter_schedule`.

    Returns:
        learning rate schedule

    See also:
        :func:`training_parameter_schedule`
    '''
    return training_parameter_schedule(lr, unit, epoch_size)



@typemap
def momentum_schedule(momentum, epoch_size=None):                                            [docs]
    '''
    Create a per-minibatch momentum schedule (using the same semantics as
    :func:`training_parameter_schedule` with the `unit=UnitType.minibatch`).

    Args:
        momentum (float or list): see parameter ``schedule`` in
         :func:`training_parameter_schedule`.
        epoch_size (int): see parameter ``epoch_size`` in
         :func:`training_parameter_schedule`.

    If you want to provide momentum values in a minibatch-size
    agnostic way, use :func:`momentum_as_time_constant_schedule`.

    Examples:
        >>> # Use a fixed momentum of 0.99 for all samples
        >>> m = momentum_schedule(0.99)

        >>> # Use the momentum value 0.99 for the first 1000 samples,
        >>> # then 0.9 for the remaining ones
        >>> m = momentum_schedule([0.99,0.9], 1000)
        >>> m[0], m[999], m[1000], m[1001]
        (0.99, 0.99, 0.9, 0.9)

        >>> # Use the momentum value 0.99 for the first 999 samples,
        >>> # then 0.88 for the next 888 samples, and 0.77 for the
        >>> # the remaining ones
        >>> m = momentum_schedule([(999,0.99),(888,0.88),(0, 0.77)])
        >>> m[0], m[998], m[999], m[999+888-1], m[999+888]
        (0.99, 0.99, 0.88, 0.88, 0.77)

    Returns:
        momentum schedule
    '''
    return training_parameter_schedule(momentum, UnitType.minibatch, epoch_size)



@typemap
def momentum_as_time_constant_schedule(momentum, epoch_size=None):                           [docs]
    '''
    Create a momentum schedule in a minibatch-size agnostic way
    (using the same semantics as :func:`training_parameter_schedule`
    with `unit=UnitType.sample`).

    Args:
        momentum (float or list): see parameter ``schedule`` in
         :func:`training_parameter_schedule`.
        epoch_size (int): see parameter ``epoch_size`` in
         :func:`training_parameter_schedule`.

    CNTK specifies momentum in a minibatch-size agnostic way as the time
    constant (in samples) of a unit-gain 1st-order IIR filter. The value
    specifies the number of samples after which a gradient has an effect of
    1/e=37%.

    If you want to specify the momentum per sample (or per minibatch),
    use :func:`momentum_schedule`.

    Examples:
        >>> # Use a fixed momentum of 1100 for all samples
        >>> m = momentum_as_time_constant_schedule(1100)

        >>> # Use the time constant 1100 for the first 1000 samples,
        >>> # then 1500 for the remaining ones
        >>> m = momentum_as_time_constant_schedule([1100, 1500], 1000)

    Returns:
        momentum as time constant schedule
    '''
    if isinstance(momentum, (cntk_py.momentum_as_time_constant_schedule)):
        return momentum

    if isinstance(momentum, (int, float)):
        if epoch_size is not None:
            warnings.warn('When providing the schedule as a number, epoch_size is ignored',
RuntimeWarning)
        return cntk_py.momentum_as_time_constant_schedule(momentum)

    if isinstance(momentum, list):
        args = [momentum] if epoch_size is None else [momentum, epoch_size]
        return cntk_py.momentum_as_time_constant_schedule(*args)

    raise ValueError(
        'momentum must be either a float or a list, not %s' % type(momentum))


# TODO figure out how to pass infty to C++ in a portable way


@typemap
def sgd(parameters, lr,                                                          [docs]
        l1_regularization_weight=0.0, l2_regularization_weight=0.0,
        gaussian_noise_injection_std_dev=0.0, gradient_clipping_threshold_per_sample=np.inf,
        gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''sgd(parameters, lr, l1_regularization_weight=0, l2_regularization_weight=0,
gaussian_noise_injection_std_dev=0, gradient_clipping_threshold_per_sample=np.inf,
gradient_clipping_with_truncation=True)
    Creates an SGD learner instance to learn the parameters. See [1] for more
    information on how to set the parameters.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the '.parameters()' method of the root
         operator.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.
    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    See also:
        [1] L. Bottou. `Stochastic Gradient Descent Tricks
        <https://www.microsoft.com/en-us/research/publication/stochastic-gradient-tricks>`_.
Neural
        Networks: Tricks of the Trade: Springer, 2012.
    '''
    _verify_learning_rate_type(lr)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.sgd_learner(parameters, lr, additional_options)



@typemap
def momentum_sgd(parameters, lr, momentum, unit_gain=default_unit_gain_value(),            [docs]
                 l1_regularization_weight=0.0, l2_regularization_weight=0.0,
                 gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
                 gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''momentum_sgd(parameters, lr, momentum, unit_gain=default_unit_gain_value(),
l1_regularization_weight=0.0, l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates a Momentum SGD learner instance to learn the parameters.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): momentum schedule.
         For additional information, please refer to the :cntkwiki:`this CNTK Wiki article
<BrainScript-SGD-Block#converting-learning-rate-and-momentum-parameters-from-other-toolkits>`.
        unit_gain: when ``True``, momentum is interpreted as a unit-gain filter. Defaults
         to the value returned by :func:`default_unit_gain_value`.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`
    '''
    _verify_learning_rate_type(lr)
    _verify_momentum_type(momentum)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.momentum_sgd_learner(parameters, lr, momentum, unit_gain,
                                        additional_options)



@typemap
def nesterov(parameters, lr, momentum, unit_gain=default_unit_gain_value(),                    [docs]
             l1_regularization_weight=0.0, l2_regularization_weight=0.0,
             gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
             gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''nesterov(parameters, lr, momentum, unit_gain=default_unit_gain_value(),
l1_regularization_weight=0, l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates a Nesterov SGD learner instance to learn the parameters. This was
    originally proposed by Nesterov [1] in 1983 and then shown to work well in
    a deep learning context by Sutskever, et al. [2].

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): momentum schedule.
         For additional information, please refer to the :cntkwiki:`this CNTK Wiki article
<BrainScript-SGD-Block#converting-learning-rate-and-momentum-parameters-from-other-toolkits>`.
        unit_gain: when ``True``, momentum is interpreted as a unit-gain filter. Defaults
         to the value returned by :func:`default_unit_gain_value`.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    See also:
        [1] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate
O(1/ sqrt(k)). Soviet Mathematics Doklady, 1983.

        [2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. `On the
        Importance of Initialization and Momentum in Deep Learning
        <http://www.cs.toronto.edu/~fritz/absps/momentum.pdf>`_.  Proceedings
        of the 30th International Conference on Machine Learning, 2013.

    '''
    _verify_learning_rate_type(lr)
    _verify_momentum_type(momentum)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient
    return cntk_py.nesterov_learner(parameters, lr, momentum, unit_gain,
                                    additional_options)


@typemap
def adadelta(parameters, lr=learning_rate_schedule(1, UnitType.sample), rho=0.95,          [docs]
epsilon=1e-8,
            l1_regularization_weight=0.0, l2_regularization_weight=0.0,
            gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
            gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''adadelta(parameters, lr, rho, epsilon, l1_regularization_weight=0,
l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates an AdaDelta learner instance to learn the parameters. See [1] for
    more information.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        rho (float): exponential smooth factor for each minibatch.
        epsilon (float): epsilon for sqrt.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    See also:
        [1]  Matthew D. Zeiler, `ADADELTA: An Adaptive Learning Rate Method
        <https://arxiv.org/pdf/1212.5701.pdf>`_.
    '''
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.ada_delta_learner(parameters, lr, rho, epsilon,
                                    additional_options)



@typemap
def adagrad(parameters, lr, need_ave_multiplier=True,                                    [docs]
            l1_regularization_weight=0.0, l2_regularization_weight=0.0,
            gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
            gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''adagrad(parameters, lr, need_ave_multiplier=True, l1_regularization_weight=0,
l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates an AdaGrad learner instance to learn the parameters. See [1] for
    more information.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        need_ave_multiplier (bool, default):
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    See also:
        [1]  J. Duchi, E. Hazan, and Y. Singer. `Adaptive Subgradient Methods
        for Online Learning and Stochastic Optimization
        <http://www.magicbroom.info/Papers/DuchiHaSi10.pdf>`_. The Journal of
        Machine Learning Research, 2011.
    '''
    _verify_learning_rate_type(lr)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.ada_grad_learner(parameters, lr, need_ave_multiplier,
                                    additional_options)



@typemap
def fsadagrad(parameters, lr, momentum, unit_gain=default_unit_gain_value(),                [docs]
              variance_momentum=momentum_as_time_constant_schedule(720000),
              l1_regularization_weight=0.0, l2_regularization_weight=0.0,
              gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
              gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''fsadagrad(parameters, lr, momentum, unit_gain=default_unit_gain_value(),
variance_momentum=momentum_as_time_constant_schedule(720000), l1_regularization_weight=0,
l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates an FSAdaGrad learner instance to learn the parameters.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): momentum schedule.
         For additional information, please refer to the :cntkwiki:`this CNTK Wiki article
<BrainScript-SGD-Block#converting-learning-rate-and-momentum-parameters-from-other-toolkits>`.
        unit_gain: when ``True``, momentum is interpreted as a unit-gain filter. Defaults
         to the value returned by :func:`default_unit_gain_value`.
        variance_momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): variance momentum schedule. Defaults
         to ``momentum_as_time_constant_schedule(720000)``.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    '''
    _verify_learning_rate_type(lr)
    _verify_momentum_type(momentum)
    _verify_momentum_type(variance_momentum)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.fsada_grad_learner(parameters, lr, momentum, unit_gain,
                                      variance_momentum, additional_options)



@typemap
def adam(parameters, lr, momentum, unit_gain=default_unit_gain_value(),                [docs]
         variance_momentum=momentum_as_time_constant_schedule(720000),
         l1_regularization_weight=0.0, l2_regularization_weight=0.0,
         gaussian_noise_injection_std_dev=0.0, gradient_clipping_threshold_per_sample=np.inf,
         gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value(), epsilon=1e-8, adamax=False):
    '''adam(parameters, lr, momentum, unit_gain=default_unit_gain_value(),
variance_momentum=momentum_as_time_constant_schedule(720000), l1_regularization_weight=0,
l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True,
epsilon=1e-8, adamax=False)
    Creates an Adam learner instance to learn the parameters. See [1] for more
    information.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): momentum schedule.
         For additional information, please refer to the :cntkwiki:`this CNTK Wiki article
<BrainScript-SGD-Block#converting-learning-rate-and-momentum-parameters-from-other-toolkits>`.
        unit_gain: when ``True``, momentum is interpreted as a unit-gain filter. Defaults
         to the value returned by :func:`default_unit_gain_value`.
        variance_momentum (output of :func:`momentum_schedule` or
:func:`momentum_as_time_constant_schedule`): variance momentum schedule. Defaults
         to ``momentum_as_time_constant_schedule(720000)``.
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.
        epsilon (float, optional): numerical stability constant,
         defaults to 1e-8
        adamax: when ``True``, use infinity-norm variance momentum update instead of L2.
Defaults
         to False

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    See also:
        [1] D. Kingma, J. Ba. `Adam: A Method for Stochastic Optimization
        <https://arxiv.org/abs/1412.6980>`_. International Conference for
        Learning Representations, 2015.
    '''
    _verify_learning_rate_type(lr)
    _verify_momentum_type(momentum)
    _verify_momentum_type(variance_momentum)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.adam_learner(parameters, lr, momentum, unit_gain,
                                variance_momentum, epsilon, adamax, additional_options)



@typemap
def rmsprop(parameters, lr,                                                          [docs]
            gamma, inc, dec, max, min,
            need_ave_multiplier=True,
            l1_regularization_weight=0.0, l2_regularization_weight=0.0,
            gaussian_noise_injection_std_dev=0.0,
gradient_clipping_threshold_per_sample=np.inf,
            gradient_clipping_with_truncation=True,
use_mean_gradient=default_use_mean_gradient_value()):
    '''rmsprop(parameters, lr, gamma, inc, dec, max, min, need_ave_multiplier=True,
l1_regularization_weight=0, l2_regularization_weight=0, gaussian_noise_injection_std_dev=0,
gradient_clipping_threshold_per_sample=np.inf, gradient_clipping_with_truncation=True)
    Creates an RMSProp learner instance to learn the parameters.

    Args:
        parameters (list of parameters): list of network parameters to tune.
         These can be obtained by the root operator's ``parameters``.
        lr (output of :func:`learning_rate_schedule`): learning rate schedule.
        gamma (float): Trade-off factor for current and previous gradients. Common value is
0.95. Should be in range (0.0, 1.0)
        inc (float): Increasing factor when trying to adjust current learning_rate. Should be
greater than 1
        dec (float): Decreasing factor when trying to adjust current learning_rate. Should be
in range (0.0, 1.0)
        max (float): Maximum scale allowed for the initial learning_rate. Should be greater
than zero and min
        min (float): Minimum scale allowed for the initial learning_rate. Should be greater
than zero
        need_ave_multiplier (bool, default ``True``):
        l1_regularization_weight (float, optional): the L1 regularization weight per sample,
         defaults to 0.0
        l2_regularization_weight (float, optional): the L2 regularization weight per sample,
         defaults to 0.0
        gaussian_noise_injection_std_dev (float, optional): the standard deviation
         of the Gaussian noise added to parameters post update, defaults to 0.0
        gradient_clipping_threshold_per_sample (float, optional): clipping threshold
         per sample, defaults to infinity
        gradient_clipping_with_truncation (bool, default ``True``): use gradient clipping
         with truncation
        use_mean_gradient (bool, default ``False``): use averaged gradient as input to learner.
         Defaults to the value returned by :func:`default_use_mean_gradient_value()`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`
    '''
    _verify_learning_rate_type(lr)
    gaussian_noise_injection_std_dev = \
        training_parameter_schedule(
            gaussian_noise_injection_std_dev, UnitType.minibatch)

    additional_options = cntk_py.AdditionalLearningOptions()
    additional_options.l1_regularization_weight = l1_regularization_weight
    additional_options.l2_regularization_weight = l2_regularization_weight
    additional_options.gaussian_noise_injection_std_dev = gaussian_noise_injection_std_dev
    additional_options.gradient_clipping_threshold_per_sample =
gradient_clipping_threshold_per_sample
    additional_options.gradient_clipping_with_truncation = gradient_clipping_with_truncation
    additional_options.use_mean_gradient = use_mean_gradient

    return cntk_py.rmsprop_learner(parameters, lr, gamma, inc, dec, max, min,
                                   need_ave_multiplier, additional_options)



@typemap
def universal(update_func, parameters):                                              [docs]
    '''
    Creates a learner which uses a CNTK function to update the parameters.

    Args:
        update_func: function that takes parameters and gradients as arguments and
         returns a :class:`~cntk.ops.functions.Function` that performs the
         desired updates. The returned function updates the parameters by
         means of containing :func:`~cntk.ops.assign` operations.
         If ``update_func`` does not contain :func:`~cntk.ops.assign` operations
         the parameters will not be updated.
        parameters (list): list of network parameters to tune.
         These can be obtained by the root operator's `parameters`.

    Returns:
        :class:`~cntk.learners.Learner`: learner instance that can be passed to
        the :class:`~cntk.train.trainer.Trainer`

    Examples:
        >>> def my_adagrad(parameters, gradients):
        ...     accumulators = [C.constant(0, shape=p.shape, dtype=p.dtype, name='accum') for p
in parameters]
        ...     update_funcs = []
        ...     for p, g, a in zip(parameters, gradients, accumulators):
        ...         accum_new = C.assign(a, g * g)
        ...         update_funcs.append(C.assign(p, p - 0.01 * g / C.sqrt(accum_new + 1e-6)))
        ...     return C.combine(update_funcs)
        ...
        >>> x = C.input_variable((10,))
        >>> y = C.input_variable((2,))
        >>> z = C.layers.Sequential([C.layers.Dense(100, activation=C.relu),
C.layers.Dense(2)])(x)
        >>> loss = C.cross_entropy_with_softmax(z, y)
        >>> learner = C.universal(my_adagrad, z.parameters)
        >>> trainer = C.Trainer(z, loss, learner)
        >>> # now trainer can be used as any other Trainer

    '''

    from .. import constant
    args, _ = utils.get_python_function_arguments(update_func)
    if len(args) != 2:
        raise ValueError('update_func must be a function that accepts two arguments
(parameters, gradients)')
    gradients = []
    for p in parameters:
        if any(dim<0 for dim in p.shape):
            raise ValueError('parameter %s has inferred dimensions. Please create the learner
after all parameter shapes have been determined'%str(p))
        gradients.append(constant(0, shape=p.shape, dtype=p.dtype, name='grad'))

    result = update_func(parameters, gradients)

    return cntk_py.universal_learner(parameters, gradients, result)
```