```
"""Optimizers for use with JAX.

This short module contains some convenient optimizer definitions, specifically
initialization and update functions, which can be used with ndarrays or
arbitrarily-nested tuple/list/dicts of ndarrays.
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import operator
import functools

import jax.numpy as np
from jax.core import pack
from jax.tree_util import tree_map, tree_multimap



def optimizer(opt_maker):
  """Decorator to make an optimizer map over tuple/list/dict containers."""
  @functools.wraps(opt_maker)
  def tree_opt_maker(*args, **kwargs):
    init_fun, update_fun = opt_maker(*args, **kwargs)

    @functools.wraps(init_fun)
    def fmapped_init_fun(x0_tree):
      return tree_map(lambda x0: pack(init_fun(x0)), x0_tree)

    @functools.wraps(update_fun)
    def fmapped_update_fun(i, grad_tree, state_tree):
      update = lambda g, state: pack(update_fun(i, g, *state))
      return tree_multimap(update, grad_tree, state_tree)

    return fmapped_init_fun, fmapped_update_fun
  return tree_opt_maker

def iterate(state_tree):
  """Extract the current iterate from an optimizer state."""
  return tree_map(lambda state: tuple(state)[0], state_tree)
get_params = iterate
```

```
# optimizers

@optimizer
def sgd(step_size):
  """Construct init and update step functions for stochastic gradient descent.

  Args:
    step_size: positive scalar, or a callable representing a step size schedule
      that maps the iteration index to positive scalar.

  Returns:
    An (init_fun, update_fun) pair.
  """
  step_size = make_schedule(step_size)
  def init_fun(x0):
    return (x0,)
  def update_fun(i, g, x):
    return (x - step_size(i) * g,)
  return init_fun, update_fun

@optimizer
def momentum(step_size, mass):
  """Construct init and update step functions for SGD with Nesterov momentum.

  Args:
    step_size: positive scalar, or a callable representing a step size schedule
      that maps the iteration index to positive scalar.

  Returns:
    An (init_fun, update_fun) pair.
  """
  step_size = make_schedule(step_size)
  def init_fun(x0):
    v0 = np.zeros_like(x0)
    return x0, v0
  def update_fun(i, g, x, velocity):
    velocity = mass * velocity - (1. - mass) * g
    x = x + step_size(i) * velocity
    return x, velocity
  return init_fun, update_fun
```

```
@optimizer
def rmsprop(step_size, gamma=0.9, eps=1e-8):
    """Construct init and update step functions for RMSProp.

    Args:
      step_size: positive scalar, or a callable representing a step size schedule
        that maps the iteration index to positive scalar.

    Returns:
      An (init_fun, update_fun) pair.
    """
    step_size = make_schedule(step_size)
    def init_fun(x0):
      avg_sq_grad = np.ones_like(x0)
      return x0, avg_sq_grad
    def update_fun(i, g, x, avg_sq_grad):
      avg_sq_grad = avg_sq_grad * gamma + g**2 * (1. - gamma)
      x = x - step_size(i) * g / (np.sqrt(avg_sq_grad) + eps)
      return x, avg_sq_grad
    return init_fun, update_fun

@optimizer
def adam(step_size, b1=0.9, b2=0.999, eps=1e-8):
    """Construct init and update step functions for Adam.

    Args:
      step_size: positive scalar, or a callable representing a step size schedule
        that maps the iteration index to positive scalar.
      b1: optional, a positive scalar value for beta_1, the exponential decay rate
        for the first moment estimates (default 0.9).
      b2: optional, a positive scalar value for beta_2, the exponential decay rate
        for the second moment estimates (default 0.999).
      eps: optional, a positive scalar value for epsilon, a small constant for
        numerical stability (default 1e-8).

    Returns:
      An (init_fun, update_fun) pair.
    """
    step_size = make_schedule(step_size)
    def init_fun(x0):
      m0 = np.zeros_like(x0)
```

```
        v0 = np.zeros_like(x0)
        return x0, m0, v0
    def update_fun(i, g, x, m, v):
        m = (1 - b1) * g + b1 * m    # First    moment estimate.
        v = (1 - b2) * (g ** 2) + b2 * v    # Second moment estimate.
        mhat = m / (1 - b1 ** (i + 1))    # Bias correction.
        vhat = v / (1 - b2 ** (i + 1))
        x = x - step_size(i) * mhat / (np.sqrt(vhat) + eps)
        return x, m, v
    return init_fun, update_fun

# learning rate schedules

def constant(step_size):
    def schedule(i):
        return step_size
    return schedule

def exponential_decay(step_size, decay_steps, decay_rate):
    def schedule(i):
        return step_size * decay_rate ** (i / decay_steps)
    return schedule

def inverse_time_decay(step_size, decay_steps, decay_rate, staircase=False):
    if staircase:
        def schedule(i):
            return step_size / (1 + decay_rate * np.floor(i / decay_steps))
    else:
        def schedule(i):
            return step_size / (1 + decay_rate * i / decay_steps)
    return schedule

def piecewise_constant(boundaries, values):
    boundaries = np.array(boundaries)
    values = np.array(values)
    if not boundaries.ndim == values.ndim == 1:
        raise ValueError("boundaries and values must be sequences")
    if not boundaries.shape[0] == values.shape[0] - 1:
        raise ValueError("boundaries length must be one longer than values length")

    def schedule(i):
```

```
        return values[np.sum(i > boundaries)]
    return schedule

def make_schedule(scalar_or_schedule_fun):
    if callable(scalar_or_schedule_fun):
        return scalar_or_schedule_fun
    elif np.ndim(scalar_or_schedule_fun) == 0:
        return constant(scalar_or_schedule_fun)
    else:
        raise TypeError(type(scalar_or_schedule_fun))
```