```
import numpy as np

"""
This file implements various first-order update rules that are
commonly used for
training neural networks. Each update rule accepts current weights and
the
gradient of the loss with respect to those weights and produces the
next set of
weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:
  - w: A numpy array giving the current weights.
  - dw: A numpy array of the same shape as w giving the gradient of
the
    loss with respect to w.
  - config: A dictionary containing hyperparameter values such as
learning rate,
    momentum, etc. If the update rule requires caching values over
many
    iterations, then config will also hold these cached values.

Returns:
  - next_w: The next point after the update.
  - config: The config dictionary to be passed to the next iteration
of the
    update rule.

NOTE: For most update rules, the default learning rate will probably
not perform
well; however the default values of the other hyperparameters should
work well
for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w
and
setting next_w equal to w.
"""


def sgd(w, dw, config=None):
    """
    Performs vanilla stochastic gradient descent.

    config format:
    - learning_rate: Scalar learning rate.
    """
    if config is None: config = {}
```

```
    config.setdefault('learning_rate', 1e-2)

    w -= config['learning_rate'] * dw
    return w, config


def sgd_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with momentum.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - velocity: A numpy array of the same shape as w and dw used to
store a moving
      average of the gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9) # set momentum to 0.9 if it
wasn't there
    v = config.get('velocity', np.zeros_like(w))   # gets velocity,
else sets it to zero.

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the momentum update formula.  Return the updated
weights
    #   as next_w, and the updated velocity as v.
    # ================================================================
#

    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v
    # next_w = np.linalg.norm(w - v[-1])

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    config['velocity'] = v

    return next_w, config

def sgd_nesterov_momentum(w, dw, config=None):
    """
```

```
    Performs stochastic gradient descent with Nesterov momentum.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - velocity: A numpy array of the same shape as w and dw used to
store a moving
      average of the gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9) # set momentum to 0.9 if it
wasn't there
    v = config.get('velocity', np.zeros_like(w))   # gets velocity,
else sets it to zero.

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the momentum update formula.  Return the updated
weights
    #   as next_w, and the updated velocity as v.
    # ================================================================
#

    v_old = v
    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v + config['momentum'] * (v - v_old)


    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    config['velocity'] = v

    return next_w, config

def rmsprop(w, dw, config=None):
    """
    Uses the RMSProp update rule, which uses a moving average of
squared gradient
    values to set adaptive per-parameter learning rates.

    config format:
    - learning_rate: Scalar learning rate.
    - decay_rate: Scalar between 0 and 1 giving the decay rate for the
```

```
squared
      gradient cache.
    - epsilon: Small scalar used for smoothing to avoid dividing by
zero.
    - beta: Moving average of second moments of gradients.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('decay_rate', 0.99)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('a', np.zeros_like(w))

    next_w = None

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement RMSProp.  Store the next value of w as next_w.  You
need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, so they can be used for future gradients.
Concretely,
    #   config['a'] corresponds to "a" in the lecture notes.
    # ================================================================
#

    beta = 0.99
    config['a'] = beta * config['a'] + (1 - beta) * dw * dw
    next_w = w - config['learning_rate'] * dw / (np.sqrt(config['a'] +
config['epsilon']))

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    return next_w, config


def adam(w, dw, config=None):
    """
    Uses the Adam update rule, which incorporates moving averages of
both the
    gradient and its square and a bias correction term.

    config format:
    - learning_rate: Scalar learning rate.
    - beta1: Decay rate for moving average of first moment of
gradient.
```

```
    - beta2: Decay rate for moving average of second moment of
gradient.
    - epsilon: Small scalar used for smoothing to avoid dividing by
zero.
    - m: Moving average of gradient.
    - v: Moving average of squared gradient.
    - t: Iteration number.
    """
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-3)
    config.setdefault('beta1', 0.9)
    config.setdefault('beta2', 0.999)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('v', np.zeros_like(w))
    config.setdefault('a', np.zeros_like(w))
    config.setdefault('t', 0)

    next_w = None

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement Adam.  Store the next value of w as next_w.  You
need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, and in config['v'] the moving average of the
    #   first moments.  Finally, store in config['t'] the increasing
time.
    # ================================================================
#

    t = config['t'] + 1
    config['v'] = config['beta1'] * config['v'] + (1 -
config['beta1']) * dw
    config['a'] = config['beta2'] * config['a'] + (1 -
config['beta2']) * dw * dw
    v_u = config['v'] / (1 - config['beta1']**t)
    a_u = config['a'] / (1 - config['beta2']**t)
    next_w = w - config['learning_rate'] * v_u / ((np.sqrt(a_u) +
config['epsilon']))

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    return next_w, config
```