optim.py

```
import numpy as np

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

"""
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
```

```
71      v = config.get('velocity', np.zeros_like(w))      # gets velocity, else sets it to zero.
72
73      # ================================================================ #
74      # YOUR CODE HERE:
75      #   Implement the momentum update formula.  Return the updated weights
76      #   as next_w, and the updated velocity as v.
77      # ================================================================ #
78      v = config['momentum'] * v - config['learning_rate'] * dw
79      next_w = w + v
80
81      # ================================================================ #
82      # END YOUR CODE HERE
83      # ================================================================ #
84
85      config['velocity'] = v
86
87      return next_w, config
88
89    def sgd_nesterov_momentum(w, dw, config=None):
90      """
91      Performs stochastic gradient descent with Nesterov momentum.
92
93      config format:
94      - learning_rate: Scalar learning rate.
95      - momentum: Scalar between 0 and 1 giving the momentum value.
96        Setting momentum = 0 reduces to sgd.
97      - velocity: A numpy array of the same shape as w and dw used to store a moving
98        average of the gradients.
99      """
100     if config is None: config = {}
101     config.setdefault('learning_rate', 1e-2)
102     config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
103     v = config.get('velocity', np.zeros_like(w))      # gets velocity, else sets it to zero.
104
105     # ================================================================ #
106     # YOUR CODE HERE:
107     #   Implement the momentum update formula.  Return the updated weights
108     #   as next_w, and the updated velocity as v.
109     # ================================================================ #
110     v_old = v
111     v = config['momentum'] * v - config['learning_rate'] * dw
112     next_w = w + v + config['momentum'] * (v - v_old)
113
114     # ================================================================ #
115     # END YOUR CODE HERE
116     # ================================================================ #
117
118     config['velocity'] = v
119
120     return next_w, config
121
122   def rmsprop(w, dw, config=None):
123     """
124     Uses the RMSProp update rule, which uses a moving average of squared gradient
125     values to set adaptive per-parameter learning rates.
126
127     config format:
128     - learning_rate: Scalar learning rate.
129     - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
130       gradient cache.
131     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
132     - beta: Moving average of second moments of gradients.
133     """
134     if config is None: config = {}
135     config.setdefault('learning_rate', 1e-2)
136     config.setdefault('decay_rate', 0.99)
137     config.setdefault('epsilon', 1e-8)
138     config.setdefault('a', np.zeros_like(w))
139
140     next_w = None
141
142     # ================================================================ #
143     # YOUR CODE HERE:
```

```
144     #   Implement RMSProp.  Store the next value of w as next_w.  You need
145     #   to also store in config['a'] the moving average of the second
146     #   moment gradients, so they can be used for future gradients. Concretely,
147     #   config['a'] corresponds to "a" in the lecture notes.
148     # ================================================================ #
149     config['a'] = config['decay_rate'] * config['a'] + (1 - config['decay_rate']) * np.multiply(dw, dw)
150     next_w = w - np.multiply(config['learning_rate'] / (np.sqrt(config['a']) + config['epsilon']), dw)
151 
152     # ================================================================ #
153     # END YOUR CODE HERE
154     # ================================================================ #
155 
156     return next_w, config
157 
158 
159 def adam(w, dw, config=None):
160     """
161     Uses the Adam update rule, which incorporates moving averages of both the
162     gradient and its square and a bias correction term.
163 
164     config format:
165     - learning_rate: Scalar learning rate.
166     - beta1: Decay rate for moving average of first moment of gradient.
167     - beta2: Decay rate for moving average of second moment of gradient.
168     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
169     - m: Moving average of gradient.
170     - v: Moving average of squared gradient.
171     - t: Iteration number.
172     """
173     if config is None: config = {}
174     config.setdefault('learning_rate', 1e-3)
175     config.setdefault('beta1', 0.9)
176     config.setdefault('beta2', 0.999)
177     config.setdefault('epsilon', 1e-8)
178     config.setdefault('v', np.zeros_like(w))
179     config.setdefault('a', np.zeros_like(w))
180     config.setdefault('t', 0)
181 
182     next_w = None
183 
184     # ================================================================ #
185     # YOUR CODE HERE:
186     #   Implement Adam.  Store the next value of w as next_w.  You need
187     #   to also store in config['a'] the moving average of the second
188     #   moment gradients, and in config['v'] the moving average of the
189     #   first moments.  Finally, store in config['t'] the increasing time.
190     # ================================================================ #
191 
192     config['t'] += 1
193     config['v'] = config['beta1'] * config['v'] + (1 - config['beta1']) * dw
194     config['a'] = config['beta2'] * config['a'] + (1 - config['beta2']) * np.multiply(dw, dw)
195     v_corr = config['v'] / (1 - (config['beta1'] ** config['t']))
196     a_corr = config['a'] / (1 - (config['beta2'] ** config['t']))
197     next_w = w - np.multiply((config['learning_rate'] / (np.sqrt(a_corr) + config['epsilon'])), v_corr)
198     # ================================================================ #
199     # END YOUR CODE HERE
200     # ================================================================ #
201 
202     return next_w, config
203 
204 
205 
206 
207 
208 
```