

```

1 import numpy as np
2
3 """
4 This code was originally written for CS 231n at Stanford University
5 (cs231n.stanford.edu). It has been modified in various areas for use in the
6 ECE 239AS class at UCLA. This includes the descriptions of what code to
7 implement as well as some slight potential changes in variable names to be
8 consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
9 permission to use this code. To see the original version, please visit
10 cs231n.stanford.edu.
11 """
12
13 """
14 This file implements various first-order update rules that are commonly used for
15 training neural networks. Each update rule accepts current weights and the
16 gradient of the loss with respect to those weights and produces the next set of
17 weights. Each update rule has the same interface:
18
19 def update(w, dw, config=None):
20
21 Inputs:
22 - w: A numpy array giving the current weights.
23 - dw: A numpy array of the same shape as w giving the gradient of the
24     loss with respect to w.
25 - config: A dictionary containing hyperparameter values such as learning rate,
26     momentum, etc. If the update rule requires caching values over many
27     iterations, then config will also hold these cached values.
28
29 Returns:
30 - next_w: The next point after the update.
31 - config: The config dictionary to be passed to the next iteration of the
32     update rule.
33
34 NOTE: For most update rules, the default learning rate will probably not perform
35 well; however the default values of the other hyperparameters should work well
36 for a variety of different problems.
37
38 For efficiency, update rules may perform in-place updates, mutating w and
39 setting next_w equal to w.
40 """
41
42
43 def sgd(w, dw, config=None):
44     """
45     Performs vanilla stochastic gradient descent.
46
47     config format:
48     - learning_rate: Scalar learning rate.
49     """
50     if config is None:
51         config = {}
52     config.setdefault('learning_rate', 1e-2)
53
54     w -= config['learning_rate'] * dw
55     return w, config
56
57
58 def sgd_momentum(w, dw, config=None):
59     """
60     Performs stochastic gradient descent with momentum.

```

```

61
62 config format:
63 - learning_rate: Scalar learning rate.
64 - momentum: Scalar between 0 and 1 giving the momentum value.
65   Setting momentum = 0 reduces to sgd.
66 - velocity: A numpy array of the same shape as w and dw used to store a moving
67   average of the gradients.
68 """
69 if config is None:
70     config = {}
71 config.setdefault('learning_rate', 1e-2)
72 # set momentum to 0.9 if it wasn't there
73 config.setdefault('momentum', 0.9)
74 # gets velocity, else sets it to zero.
75 v = config.get('velocity', np.zeros_like(w))
76
77 # ===== #
78 # YOUR CODE HERE:
79 #   Implement the momentum update formula. Return the updated weights
80 #   as next_w, and the updated velocity as v.
81 # ===== #
82 v = config['momentum'] * v - config['learning_rate'] * dw
83 next_w = w + v
84 # ===== #
85 # END YOUR CODE HERE
86 # ===== #
87
88 config['velocity'] = v
89
90 return next_w, config
91
92
93 def sgd_nesterov_momentum(w, dw, config=None):
94     """
95     Performs stochastic gradient descent with Nesterov momentum.
96
97     config format:
98     - learning_rate: Scalar learning rate.
99     - momentum: Scalar between 0 and 1 giving the momentum value.
100       Setting momentum = 0 reduces to sgd.
101     - velocity: A numpy array of the same shape as w and dw used to store a moving
102       average of the gradients.
103     """
104     if config is None:
105         config = {}
106     config.setdefault('learning_rate', 1e-2)
107     # set momentum to 0.9 if it wasn't there
108     config.setdefault('momentum', 0.9)
109     # gets velocity, else sets it to zero.
110     v = config.get('velocity', np.zeros_like(w))
111
112     # ===== #
113     # YOUR CODE HERE:
114     #   Implement the momentum update formula. Return the updated weights
115     #   as next_w, and the updated velocity as v.
116     # ===== #
117     v_old = v
118     v = config['momentum'] * v - config['learning_rate'] * dw
119     next_w = w + v + config['momentum'] * (v-v_old)
120     # ===== #

```

```

121 # END YOUR CODE HERE
122 # ===== #
123
124 config['velocity'] = v
125
126 return next_w, config
127
128
129 def rmsprop(w, dw, config=None):
130     """
131     Uses the RMSProp update rule, which uses a moving average of squared gradient
132     values to set adaptive per-parameter learning rates.
133
134     config format:
135     - learning_rate: Scalar learning rate.
136     - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
137       gradient cache.
138     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
139     - beta: Moving average of second moments of gradients.
140     """
141     if config is None:
142         config = {}
143     config.setdefault('learning_rate', 1e-2)
144     config.setdefault('decay_rate', 0.99)
145     config.setdefault('epsilon', 1e-8)
146     config.setdefault('a', np.zeros_like(w))
147
148     next_w = None
149
150     # ===== #
151     # YOUR CODE HERE:
152     # Implement RMSProp. Store the next value of w as next_w. You need
153     # to also store in config['a'] the moving average of the second
154     # moment gradients, so they can be used for future gradients. Concretely,
155     # config['a'] corresponds to "a" in the lecture notes.
156     # ===== #
157     a = config['a']
158     eps = config['epsilon']
159     beta = config['decay_rate']
160     a = beta * a + (1-beta) * dw * dw
161     next_w = w - config['learning_rate'] * dw / (np.sqrt(a) + eps)
162     config['a'] = a
163     # ===== #
164     # END YOUR CODE HERE
165     # ===== #
166
167     return next_w, config
168
169
170 def adam(w, dw, config=None):
171     """
172     Uses the Adam update rule, which incorporates moving averages of both the
173     gradient and its square and a bias correction term.
174
175     config format:
176     - learning_rate: Scalar learning rate.
177     - beta1: Decay rate for moving average of first moment of gradient.
178     - beta2: Decay rate for moving average of second moment of gradient.
179     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
180     - m: Moving average of gradient.

```

```

181 - v: Moving average of squared gradient.
182 - t: Iteration number.
183 """
184 if config is None:
185     config = {}
186     config.setdefault('learning_rate', 1e-3)
187     config.setdefault('beta1', 0.9)
188     config.setdefault('beta2', 0.999)
189     config.setdefault('epsilon', 1e-8)
190     config.setdefault('v', np.zeros_like(w))
191     config.setdefault('a', np.zeros_like(w))
192     config.setdefault('t', 0)
193
194 next_w = None
195
196 # ===== #
197 # YOUR CODE HERE:
198 # Implement Adam. Store the next value of w as next_w. You need
199 # to also store in config['a'] the moving average of the second
200 # moment gradients, and in config['v'] the moving average of the
201 # first moments. Finally, store in config['t'] the increasing time.
202 # ===== #
203 b1 = config['beta1']
204 b2 = config['beta2']
205 eps = config['epsilon']
206 a = config['a']
207 v = config['v']
208
209 config['t'] += 1
210 v = b1 * v + (1-b1) * dw
211 a = b2 * a + (1-b2) * dw * dw
212 vu = v / (1-b1**config['t'])
213 au = a / (1-b2**config['t'])
214 next_w = w - config['learning_rate'] * vu / (np.sqrt(au) + eps)
215
216 config['a'] = a
217 config['v'] = v
218 # ===== #
219 # END YOUR CODE HERE
220 # ===== #
221
222 return next_w, config
223

```