

```

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: config = {}
51     config.setdefault('learning_rate', 1e-2)
52
53     w -= config['learning_rate'] * dw
54     return w, config
55
56
57 def sgd_momentum(w, dw, config=None):
58     """
59     Performs stochastic gradient descent with momentum.
60
61     config format:
62     - learning_rate: Scalar learning rate.
63     - momentum: Scalar between 0 and 1 giving the momentum value.
64         Setting momentum = 0 reduces to sgd.
65     - velocity: A numpy array of the same shape as w and dw used to store a moving
66         average of the gradients.
67     """
68     if config is None: config = {}
69     config.setdefault('learning_rate', 1e-2)

```

```

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

```

```

139 config.setdefault('epsilon', 1e-8)
140 config.setdefault('a', np.zeros_like(w))
141
142 next_w = None
143
144 # ===== #
145 # YOUR CODE HERE:
146 # Implement RMSProp. Store the next value of w as next_w. You need
147 # to also store in config['a'] the moving average of the second
148 # moment gradients, so they can be used for future gradients. Concretely,
149 # config['a'] corresponds to "a" in the lecture notes.
150 # ===== #
151 dr = config['decay_rate']
152 a = config['a']
153 lr = config['learning_rate']
154 eps = config['epsilon']
155
156 a = dr*a + (1 - dr) * np.square(dw)
157
158 next_w = w - lr*np.multiply(1/(np.sqrt(a)+eps), dw)
159
160
161 config['a'] = a
162
163
164 # ===== #
165 # END YOUR CODE HERE
166 # ===== #
167
168 return next_w, config
169
170
171 def adam(w, dw, config=None):
172     """
173     Uses the Adam update rule, which incorporates moving averages of both the
174     gradient and its square and a bias correction term.
175
176     config format:
177     - learning_rate: Scalar learning rate.
178     - beta1: Decay rate for moving average of first moment of gradient.
179     - beta2: Decay rate for moving average of second moment of gradient.
180     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
181     - m: Moving average of gradient.
182     - v: Moving average of squared gradient.
183     - t: Iteration number.
184     """
185     if config is None: 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     lr = config['learning_rate']
204     b1 = config['beta1']
205     b2 = config['beta2']
206     eps = config['epsilon']
207     v = config['v']

```

```
208 a = config['a']
209 t = config['t']
210 t = t+1
211
212 v = b1*v + (1-b1)*dw
213 a = b2*a + (1-b2)*np.square(dw)
214
215 v_mod = 1/(1-np.power(b1, t))*v
216 a_mod = 1/(1-np.power(b2, t))*a
217
218 next_w = w - lr*np.multiply(1/(np.sqrt(a_mod)+eps), v_mod)
219
220 config['a'] = a
221 config['v'] = v
222 config['t'] = t
223 # ===== #
224 # END YOUR CODE HERE
225 # ===== #
226
227 return next_w, config
228
229
230
231
```