

```

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
9 for
10 permission to use this code. To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14 """
15 This file implements various first-order update rules that are commonly used
16 for
17 training neural networks. Each update rule accepts current weights and the
18 gradient of the loss with respect to those weights and produces the next set
19 of
20 weights. Each update rule has the same interface:
21
22 def update(w, dw, config=None):
23
24     Inputs:
25     - w: A numpy array giving the current weights.
26     - dw: A numpy array of the same shape as w giving the gradient of the
27         loss with respect to w.
28     - config: A dictionary containing hyperparameter values such as learning
29         rate,
30         momentum, etc. If the update rule requires caching values over many
31         iterations, then config will also hold these cached values.
32
33     Returns:
34     - next_w: The next point after the update.
35     - config: The config dictionary to be passed to the next iteration of the
36         update rule.
37
38     NOTE: For most update rules, the default learning rate will probably not
39     perform
40     well; however the default values of the other hyperparameters should work
41     well
42     for a variety of different problems.
43
44     For efficiency, update rules may perform in-place updates, mutating w and
45     setting next_w equal to w.
46 """
47
48 def sgd(w, dw, config=None):
49     """
50     Performs vanilla stochastic gradient descent.
51
52     config format:
53     - learning_rate: Scalar learning rate.
54     """
55     if config is None: config = {}
56     config.setdefault('learning_rate', 1e-2)
57
58     w -= config['learning_rate'] * dw
59     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 the updated velocity as v.
77     # ===== #
78
79     alpha = config.get('momentum')
80     eps = config.get('learning_rate')
81     v = alpha*v - eps*dw
82     next_w = w+v
83
84     # ===== #
85     # END YOUR CODE HERE
86     # ===== #
87
88     config['velocity'] = v
89
90     return next_w, config
91
92 def sgd_nesterov_momentum(w, dw, config=None):
93     """
94     Performs stochastic gradient descent with Nesterov momentum.
95
96     config format:
97     - learning_rate: Scalar learning rate.
98     - momentum: Scalar between 0 and 1 giving the momentum value.
99       Setting momentum = 0 reduces to sgd.
100    - velocity: A numpy array of the same shape as w and dw used to store a
moving
101    average of the gradients.
102    """
103    if config is None: config = {}
104    config.setdefault('learning_rate', 1e-2)
105    config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't
there
106    v = config.get('velocity', np.zeros_like(w)) # gets velocity, else sets
it to zero.
107
108    # ===== #

```

```

109 # YOUR CODE HERE:
110 # Implement the momentum update formula. Return the updated weights
111 # as next_w, and the updated velocity as v.
112 # ===== #
113 alpha = config.get('momentum')
114 eps = config.get('learning_rate')
115
116 v_old = v
117 v = alpha*v_old - eps*dw
118 next_w = w + v + alpha*(v - v_old)
119
120 # ===== #
121 # END YOUR CODE HERE
122 # ===== #
123
124 config['velocity'] = v
125
126 return next_w, config
127
128 def rmsprop(w, dw, config=None):
129     """
130     Uses the RMSProp update rule, which uses a moving average of squared
131     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
137     squared
138     gradient cache.
139     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
140     - beta: Moving average of second moments of gradients.
141     """
142     if config is None: 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.
155
156     Concretely,
157     # config['a'] corresponds to "a" in the lecture notes.
158     # ===== #
159
160     v = config.get('epsilon')
161     a = config.get('a')
162     lr = config.get('learning_rate')
163     beta = config.get('decay_rate')
164
165     a = beta*a + (1-beta)*(dw**2)
166     next_w = w - np.multiply(lr / (np.sqrt(a) + v), dw)
167
168     config['a'] = a

```

```

166 # ===== #
167 # END YOUR CODE HERE
168 # ===== #
169
170 return next_w, config
171
172
173 def adam(w, dw, config=None):
174     """
175     Uses the Adam update rule, which incorporates moving averages of both the
176     gradient and its square and a bias correction term.
177
178     config format:
179     - learning_rate: Scalar learning rate.
180     - beta1: Decay rate for moving average of first moment of gradient.
181     - beta2: Decay rate for moving average of second moment of gradient.
182     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
183     - m: Moving average of gradient.
184     - v: Moving average of squared gradient.
185     - t: Iteration number.
186     """
187     if config is None: config = {}
188     config.setdefault('learning_rate', 1e-3)
189     config.setdefault('beta1', 0.9)
190     config.setdefault('beta2', 0.999)
191     config.setdefault('epsilon', 1e-8)
192     config.setdefault('v', np.zeros_like(w))
193     config.setdefault('a', np.zeros_like(w))
194     config.setdefault('t', 0)
195
196     next_w = None
197
198     # ===== #
199     # YOUR CODE HERE:
200     # Implement Adam. Store the next value of w as next_w. You need
201     # to also store in config['a'] the moving average of the second
202     # moment gradients, and in config['v'] the moving average of the
203     # first moments. Finally, store in config['t'] the increasing time.
204     # ===== #
205
206     t = config.get('t') + 1
207     v = config.get('v')
208     beta1 = config.get('beta1')
209     v = beta1*v + (1-beta1)*dw
210
211     a = config.get('a')
212     beta2 = config.get('beta2')
213     a = beta2*a + (1-beta2)*(dw**2)
214
215     v_u = v / (1 - beta1**t)
216     a_u = a / (1 - beta2**t)
217
218     lr = config.get('learning_rate')
219     eps = config.get('epsilon')
220     next_w = w - (lr / (np.sqrt(a_u) + eps) * v_u)
221
222     config['a'] = a
223     config['v'] = v
224     config['t'] = t
225

```

```
226 # ===== #
227 # END YOUR CODE HERE
228 # ===== #
229
230 return next_w, config
231
```