

```

1 import numpy as np
2 import pdb
3
4
5
6
7 def affine_forward(x, w, b):
8     """
9     Computes the forward pass for an affine (fully-connected) layer.
10
11     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
12     examples, where each example x[i] has shape (d_1, ..., d_k). We will
13     reshape each input into a vector of dimension D = d_1 * ... * d_k, and
14     then transform it to an output vector of dimension M.
15
16     Inputs:
17     - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
18     - w: A numpy array of weights, of shape (D, M)
19     - b: A numpy array of biases, of shape (M,)
20
21     Returns a tuple of:
22     - out: output, of shape (N, M)
23     - cache: (x, w, b)
24     """
25
26     # ===== #
27     # YOUR CODE HERE:
28     # Calculate the output of the forward pass. Notice the dimensions
29     # of w are D x M, which is the transpose of what we did in earlier
30     # assignments.
31     # ===== #
32
33     x_reshape = x.reshape((x.shape[0], -1)) #N x D
34     out = np.dot(x_reshape, w) + b.reshape((1, b.shape[0])) #N x M
35
36     # ===== #
37     # END YOUR CODE HERE
38     # ===== #
39
40     cache = (x, w, b)
41     return out, cache
42
43
44 def affine_backward(dout, cache):
45     """
46     Computes the backward pass for an affine layer.
47
48     Inputs:
49     - dout: Upstream derivative, of shape (N, M)
50     - cache: Tuple of:
51       - x: Input data, of shape (N, d_1, ... d_k)
52       - w: Weights, of shape (D, M)
53
54     Returns a tuple of:
55     - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
56     - dw: Gradient with respect to w, of shape (D, M)
57     - db: Gradient with respect to b, of shape (M,)
58     """
59     x, w, b = cache

```

```

60 dx, dw, db = None, None, None
61
62 # ===== #
63 # YOUR CODE HERE:
64 # Calculate the gradients for the backward pass.
65 # ===== #
66
67 # dout is N x M
68 # dx should be N x d1 x ... x dk; it relates to dout through multiplication
with w, which is D x M
69 # dw should be D x M; it relates to dout through multiplication with x,
which is N x D after reshaping
70 # db should be M; it is just the sum over dout examples
71
72 x_reshape = np.reshape(x, (x.shape[0], -1)) #N x D
73 dx_reshape = np.dot(dout, w.T)
74
75 dx = np.reshape(dx_reshape, x.shape) #N x D
76 dw = np.dot(x_reshape.T, dout) #D x M
77 db = np.sum(dout.T, axis=1, keepdims=True).T #M x 1
78
79 # ===== #
80 # END YOUR CODE HERE
81 # ===== #
82
83 return dx, dw, db
84
85 def relu_forward(x):
86     """
87     Computes the forward pass for a layer of rectified linear units (ReLUs).
88
89     Input:
90     - x: Inputs, of any shape
91
92     Returns a tuple of:
93     - out: Output, of the same shape as x
94     - cache: x
95     """
96     # ===== #
97     # YOUR CODE HERE:
98     # Implement the ReLU forward pass.
99     # ===== #
100
101     out = np.maximum(x, 0)
102
103     # ===== #
104     # END YOUR CODE HERE
105     # ===== #
106
107     cache = x
108     return out, cache
109
110
111 def relu_backward(dout, cache):
112     """
113     Computes the backward pass for a layer of rectified linear units (ReLUs).
114
115     Input:
116     - dout: Upstream derivatives, of any shape
117     - cache: Input x, of same shape as dout

```

```

118
119 Returns:
120 - dx: Gradient with respect to x
121 """
122 x = cache
123
124 # ===== #
125 # YOUR CODE HERE:
126 # Implement the ReLU backward pass
127 # ===== #
128
129 # ReLU directs linearly to those > 0
130
131 dx = dout * (x > 0)
132
133 # ===== #
134 # END YOUR CODE HERE
135 # ===== #
136
137 return dx
138
139 def svm_loss(x, y):
140     """
141     Computes the loss and gradient using for multiclass SVM classification.
142
143     Inputs:
144     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class
145     for the ith input.
146     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
147     0 <= y[i] < C
148
149     Returns a tuple of:
150     - loss: Scalar giving the loss
151     - dx: Gradient of the loss with respect to x
152     """
153     N = x.shape[0]
154     correct_class_scores = x[np.arange(N), y]
155     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
156     margins[np.arange(N), y] = 0
157     loss = np.sum(margins) / N
158     num_pos = np.sum(margins > 0, axis=1)
159     dx = np.zeros_like(x)
160     dx[margins > 0] = 1
161     dx[np.arange(N), y] -= num_pos
162     dx /= N
163     return loss, dx
164
165
166 def softmax_loss(x, y):
167     """
168     Computes the loss and gradient for softmax classification.
169
170     Inputs:
171     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class
172     for the ith input.
173     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
174     0 <= y[i] < C
175

```

```
176     Returns a tuple of:
177     - loss: Scalar giving the loss
178     - dx: Gradient of the loss with respect to x
179     """
180
181     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
182     probs /= np.sum(probs, axis=1, keepdims=True)
183     N = x.shape[0]
184     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
185     dx = probs.copy()
186     dx[np.arange(N), y] -= 1
187     dx /= N
188     return loss, dx
189
```