

```

1 import numpy as np
2 import pdb
3
4 """
5 This code was originally written for CS 231n at Stanford University
6 (cs231n.stanford.edu). It has been modified in various areas for use in the
7 ECE 239AS class at UCLA. This includes the descriptions of what code to
8 implement as well as some slight potential changes in variable names to be
9 consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
10 permission to use this code. To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14
15 def affine_forward(x, w, b):
16     """
17     Computes the forward pass for an affine (fully-connected) layer.
18
19     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
20     examples, where each example x[i] has shape (d_1, ..., d_k). We will
21     reshape each input into a vector of dimension D = d_1 * ... * d_k, and
22     then transform it to an output vector of dimension M.
23
24     Inputs:
25     - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
26     - w: A numpy array of weights, of shape (D, M)
27     - b: A numpy array of biases, of shape (M,)
28
29     Returns a tuple of:
30     - out: output, of shape (N, M)
31     - cache: (x, w, b)
32     """
33
34     # ===== #
35     # YOUR CODE HERE:
36     # Calculate the output of the forward pass. Notice the dimensions
37     # of w are D x M, which is the transpose of what we did in earlier
38     # assignments.
39     # ===== #
40     shape = x.shape
41     N = shape[0]
42     D = np.prod(shape[1:])
43     reshaped_x = np.reshape(x, (N,D))
44
45     out = reshaped_x.dot(w) + b[:, np.newaxis].T
46
47     # ===== #
48     # END YOUR CODE HERE
49     # ===== #
50
51     cache = (x, w, b)
52     return out, cache
53
54
55 def affine_backward(dout, cache):
56     """
57     Computes the backward pass for an affine layer.
58
59     Inputs:
60     - dout: Upstream derivative, of shape (N, M)
61     - cache: Tuple of:
62       - x: Input data, of shape (N, d_1, ... d_k)
63       - w: Weights, of shape (D, M)
64
65     Returns a tuple of:
66     - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
67     - dw: Gradient with respect to w, of shape (D, M)
68     - db: Gradient with respect to b, of shape (M,)
69     """

```

```

70 x, w, b = cache
71 dx, dw, db = None, None, None
72
73 # ===== #
74 # YOUR CODE HERE:
75 # Calculate the gradients for the backward pass.
76 # ===== #
77 N, M = dout.shape
78 D = w.shape[0]
79
80 reshaped_x = np.reshape(x, (N,D))
81
82 db = np.sum(dout, axis=0)
83 dw = reshaped_x.T.dot(dout)
84 dx = np.reshape(dout.dot(w.T), x.shape)
85
86
87 # ===== #
88 # END YOUR CODE HERE
89 # ===== #
90
91 return dx, dw, db
92
93 def relu_forward(x):
94     """
95     Computes the forward pass for a layer of rectified linear units (ReLU).
96
97     Input:
98     - x: Inputs, of any shape
99
100    Returns a tuple of:
101    - out: Output, of the same shape as x
102    - cache: x
103    """
104    # ===== #
105    # YOUR CODE HERE:
106    # Implement the ReLU forward pass.
107    # ===== #
108    out = np.empty_like(x)
109    out[:] = x
110    out[out<0] = 0
111    # ===== #
112    # END YOUR CODE HERE
113    # ===== #
114
115    cache = x
116    return out, cache
117
118
119 def relu_backward(dout, cache):
120     """
121     Computes the backward pass for a layer of rectified linear units (ReLU).
122
123     Input:
124     - dout: Upstream derivatives, of any shape
125     - cache: Input x, of same shape as dout
126
127     Returns:
128     - dx: Gradient with respect to x
129     """
130    x = cache
131
132    # ===== #
133    # YOUR CODE HERE:
134    # Implement the ReLU backward pass
135    # ===== #
136    dx = np.empty_like(dout)
137    dx[:] = dout
138    dx[x<0] = 0

```

```

139
140 # ===== #
141 # END YOUR CODE HERE
142 # ===== #
143
144 return dx
145
146 def svm_loss(x, y):
147     """
148     Computes the loss and gradient using for multiclass SVM classification.
149
150     Inputs:
151     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
152       for the ith input.
153     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
154       0 <= y[i] < C
155
156     Returns a tuple of:
157     - loss: Scalar giving the loss
158     - dx: Gradient of the loss with respect to x
159     """
160     N = x.shape[0]
161     correct_class_scores = x[np.arange(N), y]
162     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
163     margins[np.arange(N), y] = 0
164     loss = np.sum(margins) / N
165     num_pos = np.sum(margins > 0, axis=1)
166     dx = np.zeros_like(x)
167     dx[margins > 0] = 1
168     dx[np.arange(N), y] -= num_pos
169     dx /= N
170     return loss, dx
171
172
173 def softmax_loss(x, y):
174     """
175     Computes the loss and gradient for softmax classification.
176
177     Inputs:
178     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
179       for the ith input.
180     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
181       0 <= y[i] < C
182
183     Returns a tuple of:
184     - loss: Scalar giving the loss
185     - dx: Gradient of the loss with respect to x
186     """
187
188     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
189     probs /= np.sum(probs, axis=1, keepdims=True)
190     N = x.shape[0]
191     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
192     dx = probs.copy()
193     dx[np.arange(N), y] -= 1
194     dx /= N
195     return loss, dx

```