

```

1  import numpy as np
2
3  def affine_forward(x, w, b):
4      """
5      Computes the forward pass for an affine (fully-connected) layer.
6
7      The input x has shape (N, d_1, ..., d_k) where x[i] is the ith input.
8      We multiply this against a weight matrix of shape (D, M) where
9      D = \prod_i d_i
10
11      Inputs:
12      x - Input data, of shape (N, d_1, ..., d_k)
13      w - Weights, of shape (D, M)
14      b - Biases, of shape (M,)
15
16      Returns a tuple of:
17      - out: output, of shape (N, M)
18      - cache: (x, w, b)
19      """
20      out = None
21      #####
22      # TODO: Implement the affine forward pass. Store the result in out. You      #
23      # will need to reshape the input into rows.                                #
24      #####
25      row_dim = x.shape[0]
26      col_dim = np.prod(x.shape[1:])
27      x_reshape = x.reshape(row_dim, col_dim)
28      out = np.dot(x_reshape, w) + b
29      #####
30      #                                     END OF YOUR CODE                        #
31      #####
32      cache = (x, w, b)
33      return out, cache
34
35
36  def affine_backward(dout, cache):
37      """
38      Computes the backward pass for an affine layer.
39
40      Inputs:
41      - dout: Upstream derivative, of shape (N, M)
42      - cache: Tuple of:
43        - x: Input data, of shape (N, d_1, ... d_k)
44        - w: Weights, of shape (D, M)
45
46      Returns a tuple of:
47      - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
48      - dw: Gradient with respect to w, of shape (D, M)
49      - db: Gradient with respect to b, of shape (M,)
50      """
51      x, w, b = cache
52      dx, dw, db = None, None, None
53      #####
54      # TODO: Implement the affine backward pass.                                #
55      #####
56      row_dim = x.shape[0]
57      col_dim = np.prod(x.shape[1:])
58      x2 = np.reshape(x, (row_dim, col_dim))
59
60      dx2 = np.dot(dout, w.T) # row_dim x col_dim
61      dw = np.dot(x2.T, dout) # col_dim x M
62      db = np.dot(dout.T, np.ones(row_dim)) # M x 1
63
64      dx = np.reshape(dx2, x.shape)
65      #####
66      #                                     END OF YOUR CODE                        #
67      #####

```

```

68     return dx, dw, db
69
70
71 def relu_forward(x):
72     """
73     Computes the forward pass for a layer of rectified linear units (ReLU).
74
75     Input:
76     - x: Inputs, of any shape
77
78     Returns a tuple of:
79     - out: Output, of the same shape as x
80     - cache: x
81     """
82     out = None
83     #####
84     # TODO: Implement the ReLU forward pass. #
85     #####
86     # This secures that ReLu never goes below zero
87     out = np.maximum(0, x)
88     #####
89     #                               END OF YOUR CODE #
90     #####
91     cache = x
92     return out, cache
93
94
95 def relu_backward(dout, cache):
96     """
97     Computes the backward pass for a layer of rectified linear units (ReLU).
98
99     Input:
100    - dout: Upstream derivatives, of any shape
101    - cache: Input x, of same shape as dout
102
103    Returns:
104    - dx: Gradient with respect to x
105    """
106    dx, x = None, cache
107    #####
108    # TODO: Implement the ReLU backward pass. #
109    #####
110    # This secures that ReLu never goes below zero
111    out = np.maximum(0, x)
112    out[out > 0] = 1
113    dx = out * dout
114    #####
115    #                               END OF YOUR CODE #
116    #####
117    return dx
118

```