2/4/2021 layers.py

```
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, \ldots, d_k) and contains a minibatch of N
      examples, where each example x[i] has shape (d_1, ..., d_k). We will
20
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
      Inputs:
24
25
      - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
      - w: A numpy array of weights, of shape (D, M)
26
27
      - b: A numpy array of biases, of shape (M,)
28
      Returns a tuple of:
29
30
      - out: output, of shape (N, M)
31
      - cache: (x, w, b)
32
33
34
      35
      # YOUR CODE HERE:
         Calculate the output of the forward pass. Notice the dimensions
36
37
         of w are D x M, which is the transpose of what we did in earlier
38
         assignments.
39
      40
      out = x.reshape(x.shape[0], -1) @ w + b
41
42
43
44
      45
      # END YOUR CODE HERE
      # ----- #
46
47
      cache = (x, w, b)
48
49
      return out, cache
50
51
52 def affine backward(dout, cache):
53
      Computes the backward pass for an affine layer.
54
55
56
      Inputs:
57
      - dout: Upstream derivative, of shape (N, M)
58
      - cache: Tuple of:
59
         x: Input data, of shape (N, d_1, ... d_k)
         - w: Weights, of shape (D, M)
```

localhost:4649/?mode=python 1/4

```
2/4/2021
                                   layers.py
 61
 62
      Returns a tuple of:
      - dx: Gradient with respect to x, of shape (N, d1, ..., d k)
 63
      - dw: Gradient with respect to w, of shape (D, M)
 64
      - db: Gradient with respect to b, of shape (M,)
 65
 66
      x, w, b = cache
 67
      dx, dw, db = None, None, None
 68
 69
 70
      # YOUR CODE HERE:
 71
 72
         Calculate the gradients for the backward pass.
 73
      74
 75
      # dout is N x M
 76
      # dx should be N x d1 x \dots x dk; it relates to dout through multiplication with
   w, which is D x M
 77
      \# dw should be D x M; it relates to dout through multiplication with x, which is
   N x D after reshaping
      # db should be M; it is just the sum over dout examples
 78
 79
      x_ND = x.reshape(x.shape[0], -1) #(N, D)
 80
      dx = (dout @ w.T).reshape(x.shape)
 81
 82
      dw = x_ND.T @ dout
 83
      db = np.sum(dout, axis=0)
 84
 85
 86
      87
      # END YOUR CODE HERE
 88
      89
 90
      return dx, dw, db
 91
 92
 93 def relu_forward(x):
 94
 95
      Computes the forward pass for a layer of rectified linear units (ReLUs).
 96
 97
      Input:
      - x: Inputs, of any shape
 98
 99
      Returns a tuple of:
100
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
109
      relu = lambda x: x * (x > 0)
110
      out = relu(x)
111
      112
      # END YOUR CODE HERE
113
      114
115
      cache = x
116
      return out, cache
117
118
```

localhost:4649/?mode=python 2/4

2/4/2021 layers.py

```
119 def relu_backward(dout, cache):
120
121
       Computes the backward pass for a layer of rectified linear units (ReLUs).
122
123
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
137
       # ReLU directs linearly to those > 0
138
       dx = dout * (x > 0)
139
140
       # =========== #
141
       # END YOUR CODE HERE
142
       143
144
       return dx
145
146
147 def svm_loss(x, y):
148
149
       Computes the loss and gradient using for multiclass SVM classification.
150
151
       Inputs:
152
       - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
153
          for the ith input.
       - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
154
155
          0 \leftarrow y[i] \leftarrow C
156
157
       Returns a tuple of:
158

    loss: Scalar giving the loss

159
       - dx: Gradient of the loss with respect to x
160
161
       N = x.shape[0]
162
       correct_class_scores = x[np.arange(N), y]
163
       margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
164
       margins[np.arange(N), y] = 0
       loss = np.sum(margins) / N
165
       num pos = np.sum(margins > 0, axis=1)
166
       dx = np.zeros like(x)
167
168
       dx[margins > 0] = 1
169
       dx[np.arange(N), y] -= num_pos
170
       dx /= N
171
       return loss, dx
172
173
174 def softmax_loss(x, y):
175
176
       Computes the loss and gradient for softmax classification.
177
178
       Inputs:
```

localhost:4649/?mode=python 3/4

```
2/4/2021
                                                  layers.py
         - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
 179
 180
             for the ith input.
 181
         - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
 182
             0 \leftarrow y[i] < C
 183
         Returns a tuple of:
 184
 185
         - loss: Scalar giving the loss
 186
         - dx: Gradient of the loss with respect to x
 187
 188
 189
         probs = np.exp(x - np.max(x, axis=1, keepdims=True))
 190
         probs /= np.sum(probs, axis=1, keepdims=True)
 191
         N = x.shape[0]
 192
         loss = -np.sum(np.log(probs[np.arange(N), y])) / N
 193
         dx = probs.copy()
 194
         dx[np.arange(N), y] -= 1
         dx /= N
 195
         return loss, dx
 196
 197
```

localhost:4649/?mode=python 4/4