

svm.py

```

1  import numpy as np
2  import pdb
3
4  """
5  This code was based off of code from cs231n at Stanford University, and modified for ECE C147/C247 at UCLA.
6  """
7  class SVM(object):
8
9      def __init__(self, dims=[10, 3073]):
10         self.init_weights(dims=dims)
11
12     def init_weights(self, dims):
13         """
14         Initializes the weight matrix of the SVM. Note that it has shape (C, D)
15         where C is the number of classes and D is the feature size.
16         """
17         self.W = np.random.normal(size=dims)
18
19     def loss(self, X, y):
20         """
21         Calculates the SVM loss.
22
23         Inputs have dimension D, there are C classes, and we operate on minibatches
24         of N examples.
25
26         Inputs:
27         - X: A numpy array of shape (N, D) containing a minibatch of data.
28         - y: A numpy array of shape (N,) containing training labels; y[i] = c means
29           that X[i] has label c, where 0 <= c < C.
30
31         Returns a tuple of:
32         - loss as single float
33         """
34
35         # compute the loss and the gradient
36         num_classes = self.W.shape[0]
37         num_train = X.shape[0]
38         loss = 0.0
39
40         for i in np.arange(num_train):
41             # ===== #
42             # YOUR CODE HERE:
43             # Calculate the normalized SVM loss, and store it as 'loss'.
44             # (That is, calculate the sum of the losses of all the training
45             # set margins, and then normalize the loss by the number of
46             # training examples.)
47             # ===== #
48             curr_label = y[i]
49             for j in range(num_classes):
50                 if j != curr_label:
51                     loss += max(0, 1 + self.W[j].dot(X[i]) - self.W[curr_label].dot(X[i]))
52
53             loss /= num_train
54
55             # ===== #
56             # END YOUR CODE HERE
57             # ===== #
58
59         return loss
60
61     def loss_and_grad(self, X, y):
62         """
63         Same as self.loss(X, y), except that it also returns the gradient.
64
65         Output: grad -- a matrix of the same dimensions as W containing
66                the gradient of the loss with respect to W.
67         """
68
69         # compute the loss and the gradient
70         num_classes = self.W.shape[0]
71         num_train = X.shape[0]
72         loss = 0.0
73         grad = np.zeros_like(self.W)
74
75         for i in np.arange(num_train):

```

```

76 # ===== #
77 # YOUR CODE HERE:
78 # Calculate the SVM loss and the gradient. Store the gradient in
79 # the variable grad.
80 # ===== #
81 curr_label = y[i]
82 for j in range(num_classes):
83     if j != curr_label:
84         a_j = self.W[j].dot(X[i])
85         a_y_i = self.W[curr_label].dot(X[i])
86         hinge_loss = max(0, 1 + a_j - a_y_i)
87         loss += hinge_loss
88         if hinge_loss > 0:
89             grad[curr_label] -= X[i]
90             grad[j] += X[i]
91
92 grad /= num_train
93 loss /= num_train
94
95 # ===== #
96 # END YOUR CODE HERE
97 # ===== #
98
99 return loss, grad
100
101 def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
102     """
103     sample a few random elements and only return numerical
104     in these dimensions.
105     """
106
107     for i in np.arange(num_checks):
108         ix = tuple([np.random.randint(m) for m in self.W.shape])
109
110         oldval = self.W[ix]
111         self.W[ix] = oldval + h # increment by h
112         fxph = self.loss(X, y)
113         self.W[ix] = oldval - h # decrement by h
114         fxmh = self.loss(X, y) # evaluate f(x - h)
115         self.W[ix] = oldval # reset
116
117         grad_numerical = (fxph - fxmh) / (2 * h)
118         grad_analytic = your_grad[ix]
119         rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
120         print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))
121
122 def fast_loss_and_grad(self, X, y):
123     """
124     A vectorized implementation of loss_and_grad. It shares the same
125     inputs and outputs as loss_and_grad.
126     """
127     loss = 0.0
128     grad = np.zeros(self.W.shape) # initialize the gradient as zero
129
130     # ===== #
131     # YOUR CODE HERE:
132     # Calculate the SVM loss WITHOUT any for loops.
133     # ===== #
134     num_classes = self.W.shape[0]
135     num_train = X.shape[0]
136     hinge = self.W.dot(X.T).T # Dim: (num_train, num_classes)
137     hinge_labeled = np.expand_dims(hinge[np.arange(hinge.shape[0]), y], axis=1) # Dims: (num_train, 1)
138     hinge = 1 + hinge - hinge_labeled # Dims: (num_train, num_classes)
139     hinge[np.arange(hinge.shape[0]), y] = 0
140     zeros = np.zeros(hinge.shape)
141     hinge = np.stack((hinge, zeros))
142     hinge = np.amax(hinge, axis=0)
143     loss = (1/float(num_train)) * np.sum(np.sum(hinge, axis=1), axis=0)
144     # ===== #
145     # END YOUR CODE HERE
146     # ===== #
147
148
149
150 # ===== #
151 # YOUR CODE HERE:
152 # Calculate the SVM grad WITHOUT any for loops.
153 # ===== #

```

```

154 indicators = hinge
155 indicators[hinge > 0] = 1
156 row_sum = np.sum(indicators, axis=1)
157 indicators[np.arange(num_train), y] = -row_sum.T
158 grad = X.T.dot(indicators).T
159 grad /= float(num_train)
160 # ===== #
161 # END YOUR CODE HERE
162 # ===== #
163
164 return loss, grad
165
166 def train(self, X, y, learning_rate=1e-3, num_iters=100,
167         batch_size=200, verbose=False):
168     """
169     Train this linear classifier using stochastic gradient descent.
170
171     Inputs:
172     - X: A numpy array of shape (N, D) containing training data; there are N
173         training samples each of dimension D.
174     - y: A numpy array of shape (N,) containing training labels; y[i] = c
175         means that X[i] has label 0 ≤ c < C for C classes.
176     - learning_rate: (float) learning rate for optimization.
177     - num_iters: (integer) number of steps to take when optimizing
178     - batch_size: (integer) number of training examples to use at each step.
179     - verbose: (boolean) If true, print progress during optimization.
180
181     Outputs:
182     A list containing the value of the loss function at each training iteration.
183     """
184     num_train, dim = X.shape
185     num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
186
187     self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights of self.W
188
189     # Run stochastic gradient descent to optimize W
190     loss_history = []
191
192     for it in np.arange(num_iters):
193         X_batch = None
194         y_batch = None
195
196         # ===== #
197         # YOUR CODE HERE:
198         # Sample batch_size elements from the training data for use in
199         # gradient descent. After sampling,
200         # - X_batch should have shape: (dim, batch_size)
201         # - y_batch should have shape: (batch_size,)
202         # The indices should be randomly generated to reduce correlations
203         # in the dataset. Use np.random.choice. It's okay to sample with
204         # replacement.
205         # ===== #
206         idx = np.random.randint(low=0, high=X.shape[0], size=batch_size)
207         X_batch = X[idx]
208         y_batch = y[idx]
209         # ===== #
210         # END YOUR CODE HERE
211         # ===== #
212
213         # evaluate loss and gradient
214         loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
215         loss_history.append(loss)
216
217         # ===== #
218         # YOUR CODE HERE:
219         # Update the parameters, self.W, with a gradient step
220         # ===== #
221         self.W = self.W - learning_rate * grad
222         # ===== #
223         # END YOUR CODE HERE
224         # ===== #
225
226         if verbose and it % 100 == 0:
227             print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
228
229     return loss_history
230
231 def predict(self, X):

```

```
232 """
233 Inputs:
234 - X: N x D array of training data. Each row is a D-dimensional point.
235
236 Returns:
237 - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
238   array of length N, and each element is an integer giving the predicted
239   class.
240 """
241 y_pred = np.zeros(X.shape[1])
242
243
244 # ===== #
245 # YOUR CODE HERE:
246 # Predict the labels given the training data with the parameter self.W.
247 # ===== #
248 y_pred = np.argmax(self.W.dot(X.T), axis=0)
249 # ===== #
250 # END YOUR CODE HERE
251 # ===== #
252
253 return y_pred
254
255
```