assignment1

钟嘉伦、孙昊海

KNN

```python
def compute_distances_two_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using a nested loop over both the training data and the
  test data.

  Inputs:
  - X: A numpy array of shape (num_test, D) containing test data.

  Returns:
  - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    is the Euclidean distance between the ith test point and the jth training
    point.
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  for i in range(num_test):
    for j in range(num_train):
      #####################################################################
      # TODO:                                                             #
      # Compute the l2 distance between the ith test point and the jth    #
      # training point, and store the result in dists[i, j]. You should   #
      # not use a loop over dimension.                                    #
      #####################################################################
      dists[i][j] = np.sqrt(np.sum(np.square(X[i] - self.X_train[j])))
      #####################################################################
      #                       END OF YOUR CODE                            #
      #####################################################################
  return dists
```

```python
def compute_distances_no_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    #########################################################################
    # TODO:                                                                 #
    # Compute the l2 distance between all test points and all training      #
    # points without using any explicit loops, and store the result in      #
    # dists.                                                                #
    #                                                                       #
    # You should implement this function using only basic array operations; #
    # in particular you should not use functions from scipy.                #
    #                                                                       #
    # HINT: Try to formulate the l2 distance using matrix multiplication    #
    #       and two broadcast sums.                                         #
    #########################################################################
    dists = np.sqrt(np.sum(np.square(X), axis=1, keepdims=True) +
                      -2*np.dot(X, self.X_train.T) +
                      np.transpose(np.sum(np.square(self.X_train), axis=1, keepdims=True)))
    #########################################################################
    #                         END OF YOUR CODE                              #
    #########################################################################
    return dists
```

```python
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

```python
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

Matrix derivation

布局方式

分子布局(numerator layout)

分母布局(denominator layout)，常用

假定所有的向量都是列向量，向量y对标量x求导

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

在分子布局下，

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix}$$

而在分母布局下，

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} & \cdots & \frac{\partial y_m}{\partial x} \end{bmatrix}$$

向量对向量求导，

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

标量对矩阵求导，

$$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{12}} & \cdots & \frac{\partial y}{\partial x_{1q}} \\ \frac{\partial y}{\partial x_{21}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{2q}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{p1}} & \frac{\partial y}{\partial x_{p2}} & \cdots & \frac{\partial y}{\partial x_{pq}} \end{bmatrix}$$

矩阵对标量求导，

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{21}}{\partial x} & \cdots & \frac{\partial y_{m1}}{\partial x} \\ \frac{\partial y_{12}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{m2}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{1n}}{\partial x} & \frac{\partial y_{2n}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

Softmax loss

```python
def softmax_loss_vectorized(W, X, y, reg):
  """
  Softmax loss function, vectorized version.

  Inputs and outputs are the same as softmax_loss_naive.
  """
  # Initialize the loss and gradient to zero.
  loss = 0.0
  dW = np.zeros_like(W)

  #############################################################################
  # TODO: Compute the softmax loss and its gradient using no explicit loops.  #
  # Store the loss in loss and the gradient in dW. If you are not careful     #
  # here, it is easy to run into numeric instability. Don't forget the        #
  # regularization!                                                           #
  #############################################################################
  num_train = X.shape[0]
  num_feature = W.shape[1]
  WX = np.matmul(X, W)
  out_put = np.exp(WX)/np.sum(np.exp(WX), axis=1, keepdims=True)
  loss = 1/num_train * np.sum(-np.log(out_put[range(num_train), list(y)])) + 0.5 * reg * np.sum(W*W)
  out_put[range(num_train), list(y)] -= 1
  out_put = np.matmul(X.T, out_put)
  dW = out_put/num_train + reg * W
  #############################################################################
  #                          END OF YOUR CODE                                 #
  #############################################################################

  return loss, dW
```

```python
# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
iteration 200 / 2000: loss 142.693203
iteration 300 / 2000: loss 87.006073
iteration 400 / 2000: loss 53.436460
iteration 500 / 2000: loss 33.061770
iteration 600 / 2000: loss 20.800984
iteration 700 / 2000: loss 13.418516
iteration 800 / 2000: loss 8.915467
iteration 900 / 2000: loss 6.217389
iteration 1000 / 2000: loss 4.481232
iteration 1100 / 2000: loss 3.554035
iteration 1200 / 2000: loss 2.907864
iteration 1300 / 2000: loss 2.526970
iteration 1400 / 2000: loss 2.350927
iteration 1500 / 2000: loss 2.229533
iteration 1600 / 2000: loss 2.204757
iteration 1700 / 2000: loss 2.087207
iteration 1800 / 2000: loss 2.100128
iteration 1900 / 2000: loss 2.145388
iteration 0 / 2000: loss 772.457640
```

```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.358000
```