Helper_function_implementations

January 26, 2025

1 KNN.py

[ ]:
```
import numpy as np
import pdb


class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    - norm: the function with which the norm is taken.

    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.
    """
    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))
      #norm = 2
```

```
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in np.arange(num_test):

      for j in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Compute the distance between the ith test point and the jth
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #
          distance = X[i, :] - self.X_train[j, :]
          dists[i, j] = norm(distance)

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

    return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    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))

    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #   NOT use a for loop (or list comprehension).  You may only use
    #   numpy operations.
    #
    #   HINT: use broadcasting.  If you have a shape (N,1) array and
```

```python
        #   a shape (M,) array, adding them together produces a shape (N, M)
        #   array.
        # ================================================================ #
        dists = np.sqrt(np.sum(X**2, axis = 1).reshape((num_test, 1)) +np.sum(self.
↪X_train**2, axis=1) - 2*np.dot(X,self.X_train.T))

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return dists


    def predict_labels(self, dists, k=1):
        """
        Given a matrix of distances between test points and training points,
        predict a label for each test point.

        Inputs:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          gives the distance betwen the ith test point and the jth training point.

        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in np.arange(num_test):
            # A list of length k storing the labels of the k nearest neighbors to
            # the ith test point.
            closest_y = []
            # ================================================================ #
            # YOUR CODE HERE:
            #   Use the distances to calculate and then store the labels of
            #   the k-nearest neighbors to the ith test point.  The function
            #   numpy.argsort may be useful.
            #
            #   After doing this, find the most common label of the k-nearest
            #   neighbors.  Store the predicted label of the ith training example
            #   as y_pred[i].  Break ties by choosing the smaller label.
            # ================================================================ #

            sorted = np.argsort(dists[i,:])
            closest_y = self.y_train[sorted[:k]]
            y_pred[i] = np.argmax(np.bincount(closest_y))
```

```
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```

2 Softmax.py

[ ]:
```
import numpy as np


class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the Softmax classifier.
    Note that it has shape (C, D) where C is the number of
    classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the normalized softmax loss.  Store it as the variable loss.
```

```
        #   (That is, calculate the sum of the losses of all the training
        #   set margins, and then normalize the loss by the number of
        #   training examples.)
        # ================================================================ #
        N,D = X.shape
        all_scores = np.dot(X, self.W.T)
        for i in range(N):
            score_i = all_scores[i,:]
            score_i -= np.max(score_i)
            current_score = score_i[y[i]]
            loss += current_score - np.log(np.sum(np.exp(score_i)))
        loss /= -N
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return loss

    def loss_and_grad(self, X, y):
        """
        Same as self.loss(X, y), except that it also returns the gradient.

        Output: grad -- a matrix of the same dimensions as W containing
          the gradient of the loss with respect to W.
        """

        # Initialize the loss and gradient to zero.
        loss = 0.0
        grad = np.zeros_like(self.W)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the softmax loss and the gradient. Store the gradient
        #   as the variable grad.
        # ================================================================ #
        N = X.shape[0]
        M = self.W.shape[0]
        all_scores = np.dot(X, self.W.T)
        for i in range(N):
            score_i = all_scores[i,:]
            score_i -= np.max(score_i)
            grad[y[i]] += X[i]
            for m in range(M):
                grad[m] -= np.exp(score_i[m])/np.sum(np.exp(score_i))*X[i]
        loss = self.loss(X,y)
        grad /= -N
        # ================================================================ #
```

```python
    # END YOUR CODE HERE
    # ================================================================ #


    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

    for i in np.arange(num_checks):
      ix = tuple([np.random.randint(m) for m in self.W.shape])

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +
↪abs(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
↪grad_analytic, rel_error))

  def fast_loss_and_grad(self, X, y):
    """
    A vectorized implementation of loss_and_grad. It shares the same
    inputs and ouptuts as loss_and_grad.
    """
    loss = 0.0
    grad = np.zeros(self.W.shape) # initialize the gradient as zero

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ================================================================ #
    N = X.shape[0]  # Number of samples
    C = self.W.shape[0]  # Number of classes

    # Compute scores
    scores = np.dot(X, self.W.T)  # Shape: (N, C)
    scores -= np.max(scores, axis=1, keepdims=True)  # For numerical stability
```

```
    # Compute softmax probabilities
    exp_scores = np.exp(scores)  # Shape: (N, C)
    probabilities = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)  #␣
↪Shape: (N, C)

    # Compute loss
    correct_log_probs = -np.log(probabilities[np.arange(N), y])  # Shape: (N,)
    loss = np.sum(correct_log_probs) / N  # Scalar loss

    # Compute gradient
    probabilities[np.arange(N), y] -= 1  # Subtract 1 for correct class␣
↪probabilities
    grad = np.dot(probabilities.T, X) / N  # Shape: (C, D)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c
      means that X[i] has label 0 <= c < C for C classes.
    - learning_rate: (float) learning rate for optimization.
    - num_iters: (integer) number of steps to take when optimizing
    - batch_size: (integer) number of training examples to use at each step.
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is␣
↪number of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]])        # initializes␣
↪the weights of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []
```

```python
    for it in np.arange(num_iters):
        X_batch = None
        y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #     gradient descent.  After sampling,
      #     - X_batch should have shape: (batch_size, dim)
      #     - y_batch should have shape: (batch_size,)
      #   The indices should be randomly generated to reduce correlations
      #   in the dataset.  Use np.random.choice.  It's okay to sample with
      #   replacement.
      # ================================================================ #
        indexes = np.random.choice(num_train, batch_size)
        X_batch = X[indexes]
        y_batch = y[indexes]
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #
      # evaluate loss and gradient
        loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
        loss_history.append(loss)

      # ================================================================ #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ================================================================ #
        self.W -= learning_rate*grad

      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #
        if verbose and it % 100 == 0:
            print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
```

```
        class.
    """
    y_pred = np.zeros(X.shape[1])
    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ================================================================ #
    all_scores = np.dot(X, self.W.T)
    all_scores = (all_scores.T - np.max(all_scores, axis = 1)).T
    probabilities = np.exp(all_scores)/np.sum(np.exp(all_scores), axis = 1,␣
↪keepdims = True)
    y_pred = np.argmax(probabilities, axis = 1) #y_pred should be a 1 dim array␣
↪of length N
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```