Related Python Code

1 KNN

```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University,
and modified for ECE C147/C247 at UCLA.
"""

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]
```

```python
    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].
    # ================================================================ #

        dists[i,j] = norm(X[i] - self.X_train[j])

    # ================================================================ #
    # 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
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ================================================================ #

    xsq = np.square(X).sum(axis=1).reshape(num_test, 1)
    ysq = np.square(self.X_train).sum(axis=1)
    xy2 = 2*np.dot(X, (self.X_train.T))
    dists = np.sqrt(xsq + ysq - xy2)

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

```python
    # ================================================================ #

    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.
      # ================================================================ #
      idx = np.argsort(dists[i,:])
      near_x = idx[0:k]
      near_y = self.y_train[near_x]
      max_idx = np.argmax(np.bincount(near_y))
      y_pred[i] = np.amin(max_idx) # breaking ties by selecting the minimal index

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

    return y_pred
```

2 SVM

```python
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University,
and modified for ECE C147/C247 at UCLA.
"""


class SVM(object):

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

    def init_weights(self, dims):
        """
            Initializes the weight matrix of the SVM.  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)

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

        # compute the loss and the gradient
        num_classes = self.W.shape[0]
        num_train = X.shape[0]
        loss = 0.0

        for i in np.arange(num_train):
            # ================================================================ #
            # YOUR CODE HERE:
            #   Calculate the normalized SVM loss, and store it as '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.)
            # ================================================================ #
            for j in range(num_classes):
```

```
                z_j = 0
                if y[i] != j:
                    z_j = 1 + self.W[j].T.dot(X[i]) - self.W[y[i]].T.dot(X[i])
                    loss += max(0, z_j)
        loss /= num_train
        # ================================================================ #
        # 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.
            """

        # compute the loss and the gradient
        num_classes = self.W.shape[0]
        num_train = X.shape[0]
        loss = 0.0
        grad = np.zeros_like(self.W)

        for i in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM loss and the gradient.  Store the gradient in
        #   the variable grad.
        # ================================================================ #
        # get loss
            for j in range(num_classes):
                z_j = 0
                if y[i] != j:
                    z_j = 1 + self.W[j].T.dot(X[i]) - self.W[y[i]].T.dot(X[i])
                    loss += max(0, z_j)
                    # get gradients
                    grad[j] += X[i] * (z_j > 0)
                    grad[y[i]] -= X[i] * (z_j > 0)

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

        loss /= num_train
        grad /= num_train

        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
```

```python
        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 SVM loss WITHOUT any for loops.
        # ================================================================ #
        num_train = X.shape[0]
        aj = np.dot(X, self.W.T)
        ay = aj[np.arange(num_train), y]
        ay = np.resize(ay, (num_train, 1))

        # after this, the loss of ay will become 1
        z = np.maximum(0, 1 + aj - ay)
        z[np.arange(num_train), y] = 0
        loss = np.sum(z)/num_train
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM grad WITHOUT any for loops.
        # ================================================================ #
        z[z > 0] = 1
        z[np.arange(num_train), y] = -np.sum(z, axis=1)
        grad = np.dot(z.T, X) / num_train
        # ================================================================ #
```

```python
    # 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
    # assume y takes values 0...K-1 where K is number of classes
    num_classes = np.max(y) + 1

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

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

    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: (dim, batch_size)
        #     - 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.
        # ================================================================ #
        idx = np.random.choice(len(X), size=batch_size, replace=False)
        X_batch = X[idx]
        y_batch = y[idx]
        # print(X_batch.shape, y_batch.shape)
        # ================================================================ #
        # 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 with the parameter self.W.
        # ================================================================ #
        y_pred = np.argmax(np.dot(self.W, X.T), axis=0)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```

3 Softmax

```python
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.)
        # ================================================================ #

        num_classes = self.W.shape[0]
        num_train = X.shape[0]

        for i in np.arange(num_train):
            ayi = self.W[y[i]].dot(X[i])
            sum_aj = 0
            for j in range(num_classes):
                sum_aj += np.exp(np.dot(self.W[j], X[i]))
```

```
            loss += (np.log(sum_aj) - ayi)
        loss /= num_train

        # ================================================================ #
        # 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.
        # ================================================================ #
        num_classes = self.W.shape[0]
        num_train = X.shape[0]

        for i in np.arange(num_train):
            ayi = self.W[y[i]].dot(X[i])
            sum_aj = 0
            for j in range(num_classes):
                sum_aj += np.exp(np.dot(self.W[j], X[i]))
            loss += (np.log(sum_aj) - ayi)
            # calculate gradient
            for j in range(num_classes):
                aj = np.exp(np.dot(self.W[j], X[i]))
                grad[j] -= X[i] * ((j == y[i]) - aj / sum_aj)
        loss /= num_train
        grad /= num_train

        # ================================================================ #
        # 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.
    # ================================================================ #
    num_train = X.shape[0]

    aj = np.dot(X, self.W.T)
    # looks like python just handles the loss fine without normalization
    # log_k = - np.max(aj, axis=1)
    ej = np.exp(aj)
    ey = ej[np.arange(num_train), y]
    sum_ej = np.sum(ej, axis=1)
    loss = -np.sum(np.log(ey/sum_ej))/num_train

    # Gradients
    probs = ej / np.sum(ej, axis=1, keepdims=True)
    probs[np.arange(num_train), y] -= 1
    grad = np.dot(probs.T, X)/num_train
    # ================================================================ #
    # 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
    # assume y takes values 0...K-1 where K is number of classes
    num_classes = np.max(y) + 1

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

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

    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: (dim, batch_size)
        #     - 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.
        # ================================================================ #
        idx = np.random.choice(len(X), size=batch_size, replace=False)
        X_batch = X[idx]
        y_batch = y[idx]
        # ================================================================ #
        # 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.
        # ================================================================ #
        y_pred = np.argmax(np.dot(self.W, X.T), axis=0)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```