04_Neural_Networks\cattern\neural_net.py

```
from __future__ import print_function

import numpy as np
import matplotlib.pyplot as plt

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (D, H)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (H, C)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(input_size, hidden_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(hidden_size, output_size)
    self.params['b2'] = np.zeros(output_size)

  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
```

```
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optional; if it
      is not passed then we only return scores, and if it is passed then we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function; has the same keys as self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None
    ##############################################################################
    # T#5: Perform the forward pass, computing the class scores for the      #
    # input.                                                                    #
    # Store the result in the scores variable, which should be an array of      #
    # shape (N, C). Note that this does not include the softmax                 #
    # HINT: This is just a series of matrix multiplication.                     #
    ##############################################################################
    h1 = X @ W1 + b1
    h1_relu = np.maximum(0, h1)
    scores = h1_relu @ W2 + b2
    ##############################################################################
    #                              END OF T#5                                #
    ##############################################################################

    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None
    ##############################################################################
    # T#6: Finish the forward pass, and compute the loss. This should include#
    # both the data loss and L2 regularization for W1 and W2. Store the result  #
    # in the variable loss, which should be a scalar. Use the Softmax           #
    # classifier loss.                                                          #
    ##############################################################################
```

```
    shift_scores = scores - np.max(scores, axis=1, keepdims=True) # prevent overflow
    softmax = np.exp(shift_scores)
    divider = np.sum(softmax, axis=1, keepdims=True)
    softmax = softmax / divider

    loss = -np.sum(np.log(softmax[np.arange(N), y]))

    # Average for each data
    loss /= y.shape[0]
    # Regularization 0.5 * lambda * sigma (every weight square)
    loss += 0.5 * reg * (np.sum(W1**2) + np.sum(W2**2))
    #############################################################################
    #                              END OF T#6                                   #
    #############################################################################

    # Backward pass: compute gradients
    grads = {}
    #############################################################################
    # T#7: Compute the backward pass, computing derivatives of the weights    #
    # and biases. Store the results in the grads dictionary. For example,       #
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    # don't forget about the regularization term                                #
    #############################################################################

    # init with gradient of regularization term
    grads["W1"] = reg * self.params["W1"]
    grads["W2"] = reg * self.params["W2"]
    grads["b1"] = np.zeros(b1.shape)
    grads["b2"] = np.zeros(b2.shape)

    dscores = softmax.copy()
    dscores[np.arange(N), y] -= 1
    dscores /= N  # if not divide here then divide at the the last step of calculation which doing it here is better

    grads["b2"] += 1 * np.sum(dscores, axis=0)

    grads["W2"] += h1_relu.T @ dscores

    dh1_relu = dscores @ W2.T
    dh1_relu[h1 <= 0] = 0
    grads["b1"] += np.sum(dh1_relu * 1, axis=0)

    grads["W1"] += X.T @ dh1_relu

    #############################################################################
    #                              END OF T#7                                   #
    #############################################################################

    return loss, grads
```

```python

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=5e-6, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in range(num_iters):
      X_batch = None
      y_batch = None

      #########################################################################
      # T#8: Create a random minibatch of training data and labels, storing#
      # them in X_batch and y_batch respectively.                             #
      # You might find np.random.choice() helpful.                            #
      #########################################################################
      batch_index = np.random.choice(num_train, batch_size, replace=True)
      X_batch = X[batch_index]
      y_batch = y[batch_index]
      #########################################################################
      #                             END OF YOUR T#8                        #
      #########################################################################

      # Compute loss and gradients using the current minibatch
      loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
      loss_history.append(loss)

```

```
      #########################################################################
      # T#9: Use the gradients in the grads dictionary to update the       #
      # parameters of the network (stored in the dictionary self.params)      #
      # using stochastic gradient descent. You'll need to use the gradients   #
      # stored in the grads dictionary defined above.                         #
      #########################################################################
      self.params['W1'] -= learning_rate * grads["W1"]
      self.params['W2'] -= learning_rate * grads["W2"]
      self.params['b1'] -= learning_rate * grads["b1"]
      self.params['b2'] -= learning_rate * grads["b2"]
      #########################################################################
      #                             END OF YOUR T#9                        #
      #########################################################################

      if verbose and it % 100 == 0:
        print('iteration %d / %d: loss %f' % (it, num_iters, loss))

      # Every epoch, check train and val accuracy and decay learning rate.
      if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        #######################################################################
        # T#10: Decay learning rate (exponentially) after each epoch        #
        #######################################################################
        learning_rate *= learning_rate_decay
        #######################################################################
        #                             END OF YOUR T#10                      #
        #######################################################################


    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }

  def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.
```

```python

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """
    y_pred = None

    ###########################################################################
    # T#11: Implement this function; it should be VERY simple!              #
    ###########################################################################
    h1_relu = np.maximum(0, X @ self.params['W1'] + self.params['b1'])
    scores = h1_relu @ self.params['W2'] + self.params['b2']
    y_pred = np.argmax(scores, axis=1)
    ###########################################################################
    #                              END OF YOUR T#11                          #
    ###########################################################################

    return y_pred
```