In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dim
ension 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 regularizat
ion on the
  weight matrices. The network uses a ReLU nonlinearity after the firs
t fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softm
ax

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

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

```
    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    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(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_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 neu
ral
    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 ea
ch y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optiona
l; if it
      is not passed then we only return scores, and if it is passed th
en 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.par
ams.
    """
    # 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

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the output scores of the neural network.  The result
    #   should be (C, N). As stated in the description for this class,
    #   there should not be a ReLU layer after the second FC layer.
    #   The output of the second FC layer is the output scores. Do not
    #   use a for loop in your implementation.
    # ================================================================
#
#    print(N,D)
    #  input - fully connected layer - ReLU - fully connected layer -
softmax
    #first layer
    HL1_pre_activation = X.dot(W1.T) + b1
    HL1_output = np.maximum(0, HL1_pre_activation) #relu

    #second layer
    HL2_pre_activation = HL1_output.dot(W2.T) + b2

    scores = HL2_pre_activation

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


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

    # Compute the loss
    loss = 0.0

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the loss of the neural network.  This includes the
    #   softmax loss and the L2 regularization for W1 and W2. Store th
e
    #   total loss in the variable loss.  Multiply the regularization
```

```
    #   loss by 0.5 (in addition to the factor reg).
    # ================================================================
#

    # scores is num_examples by num_classes
    #Loss is made up of standard softmax loss and L2 regularization
    #Generate probability of being in a class based on output (softmax
)
    class_probabilities = np.exp(scores)/np.sum(np.exp(scores), axis=1
, keepdims=True)

    """
    There will be N rows, where each row corresponds to an input.
    There are D columns, where each column will correspond to probabil
ity of being in that class.
    y is our gnd truth, so for some y=j and example i, we want class_p
robabilities[i, y=j]
    """
#    print(y)
 #   print(class_probabilities)
#    print(range[N])
    prob_of_correct_y = class_probabilities[np.arange(N), y]
    log_loss = -np.log(prob_of_correct_y)
    sum_log_loss = np.sum(log_loss)
    #divide by num examples
    loss = sum_log_loss/N

    """
    L2 regularization for matrix involves Frobenius norm.
    reg = 0.5*|| w ||_F ^2
    Frobenius norm is equiv to Sigma_iSigma_j(w_ij)^2, so we can just
do a dual sum
    """
    frob_norm_w1 = np.sum(W1**2)
    frob_norm_w2 = np.sum(W2**2)
    reg_w1 = 0.5*reg*frob_norm_w1
    reg_w2 = 0.5*reg*frob_norm_w2

    regularized_loss = reg_w1 + reg_w2
    loss += regularized_loss


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

    grads = {}
```

```
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the backward pass.  Compute the derivatives of the
    #   weights and the biases.  Store the results in the grads
    #   dictionary.  e.g., grads['W1'] should store the gradient for
    #   W1, and be of the same size as W1.
    # ================================================================
#
    """
    Source: CS231n online
    Gradient of L_i = -log(p_yi) is p_k-1 for (y_i = k)

    For weights we do a mult between the previous layer output and the
update

    We will multiply by the negative learning rate so a weight "decrea
se" at an intermediate step
    is really a weight increase.
    """



    #Calculate how we should update the scores
    update_scores = class_probabilities
    #Since we made update scores matrix by looking for only cases wher
e y_i = k, we can subtract
    #from the whole thing
#     update_scores -= np.ones_like(update_scores)
    update_scores[np.arange(N), y] -=1
    update_scores /= N

#     print(update_scores)
    #backprop W2 take gradient of output and multiply by weight matirx
    grads['W2'] = np.dot(HL1_output.T, update_scores).T

    #we want to increase the value of the activation of correct classi
fications
    grads['b2'] = np.sum(update_scores, axis=0)#, keepdims=True)

    # dL/dW2 = dL/dOut * dOut/dW2
    dHL2 = np.dot(update_scores, W2)

    # I(a>0)*dl/dh (where h is output of relu layer)
    # a in this case is HL1_pre_activation
    dLdA = dHL2
    dLdA[HL1_output <= 0] = 0


    #back prop DlDa into w and b
    grads['W1'] = np.dot(dLdA.T, X)
```

```
    grads['b1'] = np.sum(dLdA, axis=0)#, keepdims=True)


    grads['W2'] += reg * W2
    grads['W1'] += reg * W1




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

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, 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 m
eans 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 lear
ning 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 np.arange(num_iters):
      X_batch = None
```

```
      y_batch = None

      # ================================================================
== #
      # YOUR CODE HERE:
      #   Create a minibatch by sampling batch_size samples randomly.
      # ================================================================
== #
      rand_indices = np.random.choice(np.arange(num_train), batch_size
)
      X_batch = X[rand_indices]
      y_batch = y[rand_indices]
      # ================================================================
== #
      # END YOUR CODE HERE
      # ================================================================
== #

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

      # ================================================================
== #
      # YOUR CODE HERE:
      #   Perform a gradient descent step using the minibatch to updat
e
      #   all parameters (i.e., W1, W2, b1, and b2).
      # ================================================================
== #

      self.params['W2'] += -learning_rate * grads['W2']
      self.params['W1'] += -learning_rate * grads['W1']

#      print(self.params['b2'].shape, grads['b2'].shape)
      self.params['b2'] += -learning_rate * grads['b2']
      self.params['b1'] += -learning_rate * grads['b1']

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

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

      # Every epoch, check train and val accuracy and decay learning r
ate.
```

```
      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
        learning_rate *= learning_rate_decay

    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 label
s 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 poi
nts to
      classify.

    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 p
redicted
      to have class c, where 0 <= c < C.

    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    b2: Second layer biases; has shape (C,)
    """
    num_examples = X.shape[0]
    y_pred = np.empty((num_examples,), dtype=int)

    # ================================================================
#
    # YOUR CODE HERE:
    #   Predict the class given the input data.
    # ================================================================
```

```
#
    #do a forward pass for prediction
    HL1_input = np.dot(X, self.params['W1'].T) + self.params['b1']

    #apply RELU
    HL1_output = np.maximum(0, HL1_input)

    #second layer
    HL2_output = np.dot(HL1_output, self.params['W2'].T) + self.params
['b2']

    #apply softmax
    softmax = np.exp(HL2_output)/np.sum(np.exp(HL2_output), axis=1, ke
epdims=True)

    #index_of_max = np.argmax(softmax)
#    print(softmax.shape)
    for i in range(num_examples):
      max_index = np.argmax(softmax[i])
      y_pred[i] = max_index




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

    return y_pred
```