two_layer_nn

February 5, 2021

0.1 This is the 2-layer neural network workbook for ECE 247 Assignment #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a two layer neural network.

[1]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

0.2 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass

[23]:
```
from nndl.neural_net import TwoLayerNet
```

[24]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5
```

```
def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

0.2.1 Compute forward pass scores

[25]:
```
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
```

```
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231204052648e-08
```

0.2.2 Forward pass loss

[32]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
(4, 10) (10, 4)
(10, 3) (3, 10)
Difference between your loss and correct loss:
0.0
```

[33]:
```
print(loss)
```

```
1.071696123862817
```

0.2.3 Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

[41]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
 ↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
 ↪verbose=False)
    print('{} max relative error: {}'.format(param_name,␣
 ↪rel_error(param_grad_num, grads[param_name])))
```

```
W1 max relative error: 4.887032981703474e-10
W2 max relative error: 6.529301200455356e-09
b1 max relative error: 1.0
b2 max relative error: 1.0
```

0.2.4 Training the network

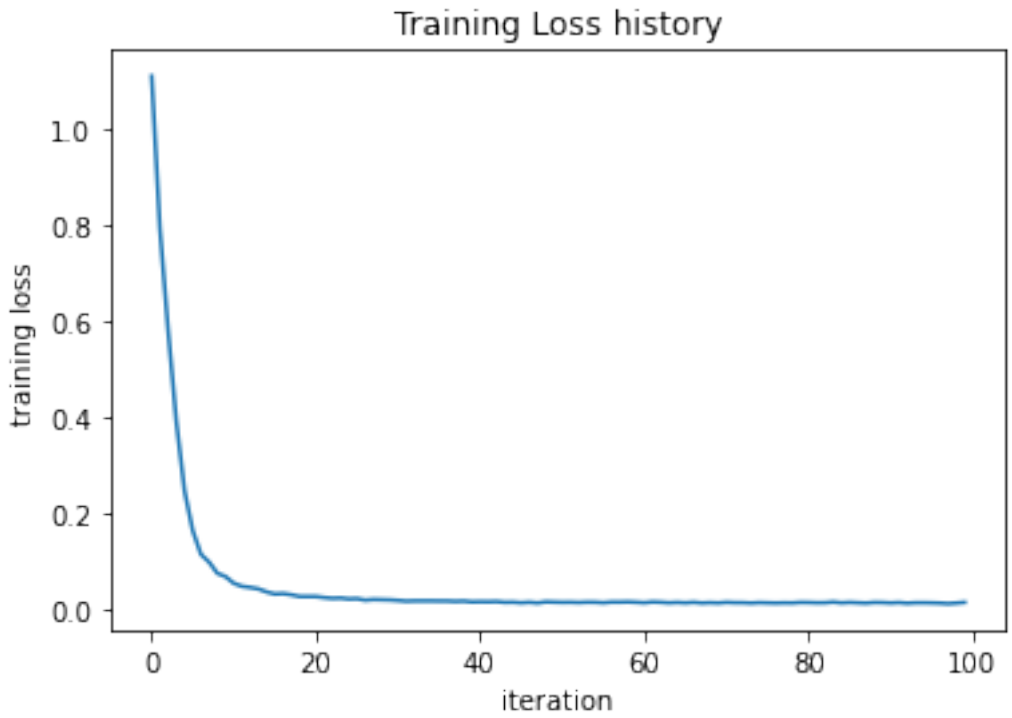
Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

[43]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.014497864587765886
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[47]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'C:/Users/Ashwin/Desktop/UCLA/current classes/ece 247/hw2/
 ↪hw2_code/cifar-10-batches-py/'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
```

```
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

0.3.1 Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

[51]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

# Save this net as the variable subopt_net for later comparison.
subopt_net = net
```

```
iteration 0 / 1000: loss 2.302784847884712
iteration 100 / 1000: loss 2.302315461263432
iteration 200 / 1000: loss 2.2984950279576903
iteration 300 / 1000: loss 2.2609952052865006
iteration 400 / 1000: loss 2.1919204730853625
iteration 500 / 1000: loss 2.087830091640278
iteration 600 / 1000: loss 2.1128624051097837
iteration 700 / 1000: loss 2.0857785427906013
iteration 800 / 1000: loss 1.9635974927174773
iteration 900 / 1000: loss 1.9971271876039314
Validation accuracy:  0.281
```

0.4 Questions:

The training accuracy isn't great.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

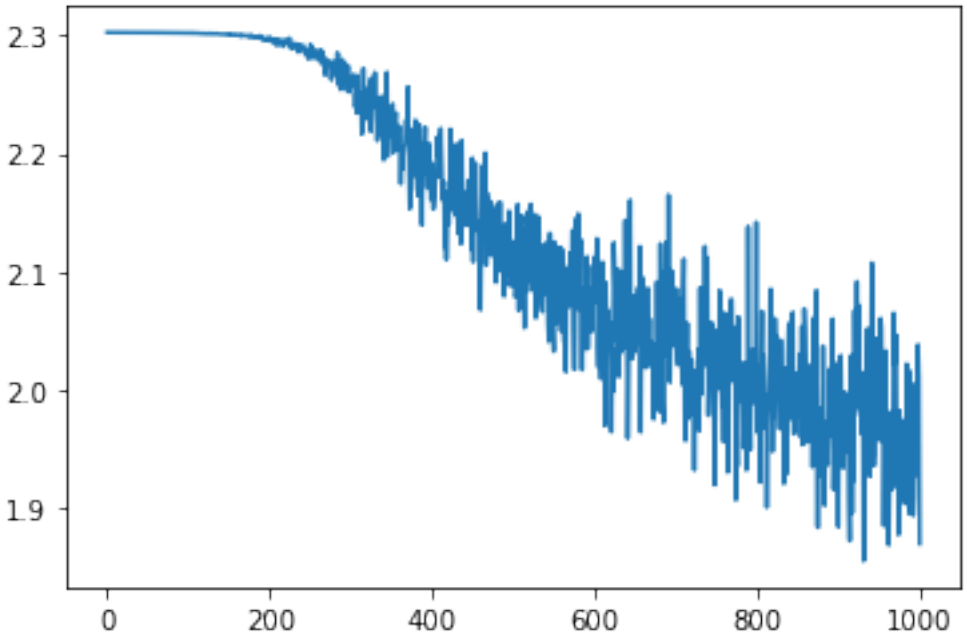
[52]:
```
stats['train_acc_history']
```

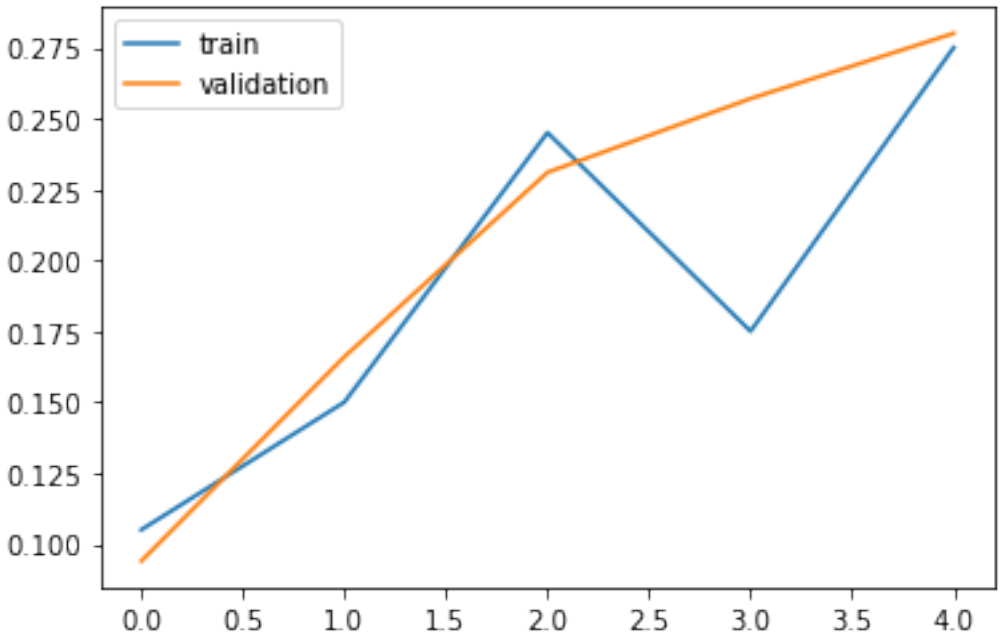
[52]:
```
[0.105, 0.15, 0.245, 0.175, 0.275]
```

[56]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies

#loss function
plt.plot(stats['loss_history'])
plt.show()
#training / validation accuracies
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='validation')
plt.legend()
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```





0.5 Answers:

(1) Based on the first graph, it looks like the gradient descent hasn't reached a minimum yet. Hence, the number of iterations are not enough it seems to settle on the optimal value.

(2) Optimize the hyperparameters, on Piazza it said we could change: num_itr, batch size, learning rate, weight decay, num_hidden_layers.

0.6 Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

[ ]:
```
import time
start = time.time()
# best_net = None # store the best model into this

# # ================================================================ #
# # YOUR CODE HERE:
# #   Optimize over your hyperparameters to arrive at the best neural
# #   network.  You should be able to get over 50% validation accuracy.
# #   For this part of the notebook, we will give credit based on the
# #   accuracy you get.  Your score on this question will be multiplied by:
# #      min(floor((X - 28%)) / %22, 1)
# #   where if you get 50% or higher validation accuracy, you get full
```

```
# #   points.
# #
# #   Note, you need to use the same network structure (keep hidden_size = 50)!
# # ================================================================ #

# op_val, op_batch, op_rate = 0, 0, 0


# #hyperparameters

# learning_rate_values = list(np.arange(-4, -3, 0.1)) #i tried [-10, -9, ..,␣
 ↪-1] but kept getting overflow
# batch_size_values = [200, 210, 220, 230, 240, 250]

# print(learning_rate_values, batch_size_values)

# for batch in batch_size_values:
#     for rate in learning_rate_values:
#         print("testing")
#         network = TwoLayerNet(input_size, hidden_size, num_classes) #keep␣
 ↪these the same, increase num_iters to 1000 since we want to see possible␣
 ↪convergence
#         network.train(X_train, y_train, X_val, y_val, num_iters=1000,␣
 ↪batch_size=batch, learning_rate=rate, learning_rate_decay=0.95, reg=1e-5,␣
 ↪verbose=False)

#         val = (network.predict(X_val) == y_val).mean()

#         print("val acc:", val, "batch_size:", batch, "learning rate:", rate)

#         if val > op_val:
#             op_val, op_batch, op_rate = val, batch, rate
#             net = network

# print("best net:")
# print("val acc:", op_val, "batch_size:", op_batch, "learning rate:", op_rate)


# # ================================================================ #
# # END YOUR CODE HERE
# # ================================================================ #
# best_net = net

best_net = None # store the best model into this

# ================================================================ #
# YOUR CODE HERE:
```

```
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied by:
#      min(floor((X - 28%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size = 50)!
# ================================================================ #

op_val, op_batch, op_rate = 0, 0, 0


#hyperparameters

learning_rate_values = [0.0004] #kept getting overflow when I tried other values
#batch_size_values = [200, 210, 220, 230, 240, 250]
batch_size_values = [210, 220, 230]
print(learning_rate_values, batch_size_values)

for batch in batch_size_values:


    for rate in learning_rate_values:
        print("testing")
        network = TwoLayerNet(input_size, hidden_size, num_classes) #keep these␣
↪the same, increase num_iters to 1000 since we want to see possible␣
↪convergence
        network.train(X_train, y_train, X_val, y_val, num_iters=1000,␣
↪batch_size=batch, learning_rate=rate, learning_rate_decay=0.95, reg=0.25,␣
↪verbose=True)

        val = (network.predict(X_val) == y_val).mean()

        print("val acc:", val, "batch_size:", batch, "learning rate:", rate)

        if val > op_val:
            op_val, op_batch, op_rate = val, batch, rate
            net = network

print("best net:")
print("val acc:", op_val, "batch_size:", op_batch, "learning rate:", op_rate)


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

```
# ================================================================ #
best_net = net

end = time.time()
print(end - start)
```

[ ]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```

0.7 Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

0.8 Answer:

Note: i was unable to optimize the hyperparameters as best as I wanted, since my computer would lag out, and even after waiting 30 minutes, my neural network would not train in that time. Hence, I couldn't test the hyperparameters, hopefully that will not affect my final grade on this part, since my computer is pretty slow. That's why I am unable to answer the questions below.

(1) I'm guessing the pictures in the best net will be clearer. However, I was not able to evaluate, hence, I'm not sure. I wrote the code assuming best_net was calculated, below, so hopefully I still get most of the points since I worked pretty hard on this section.

0.9 Evaluate on test set

[ ]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

[ ]:
```
#neural_net.py
import numpy as np
import matplotlib.pyplot as plt
from .layers import *

"""
```

```
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 to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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 (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 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

    # ============================================================== #
    # YOUR CODE HERE:
        #   Calculate the output scores of the neural network.  The result
        #   should be (N, C). 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.
    # ============================================================== #

    #our activation function is RELU based on the description
    f = lambda x : x * (x > 0) #relu activation function
    h = f(np.dot(X, W1.T) + b1)
    scores = np.dot(h, W2.T) + b2 #W2.T is basically voodoo programming but␣
↪whatever

    # ============================================================== #
```

```
    # 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 = None

    # ============================================================ #
    # YOUR CODE HERE:
        #   Calculate the loss of the neural network.  This includes the
        #         softmax loss and the L2 regularization for W1 and W2. Store␣
↪the
        #        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

    softmax_loss_var, dscores = softmax_loss(scores, y) #use scores, not X,␣
↪since this is NN, not softmax!!
    reg_loss = 0.5*reg*np.sum(W1*W1) + 0.5*reg*np.sum(W2*W2)
    loss = softmax_loss_var + reg_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.
        # ============================================================ #

    #using https://cs231n.github.io/neural-networks-case-study/ as a resource

    #first backprop gradient (dscores) into W2 and b2

    dW2 = np.dot(h.T, dscores)
    db2 = np.sum(dscores, axis=0, keepdims=True)
```

```
    #then, backprop into hidden layer
    dhidden = np.dot(dscores, W2)

    #backprop RELU
    dhidden[h <= 0] = 0

    #backprop into W and b
    dW = np.dot(X.T, dhidden)
    db = np.sum(dhidden, axis=0, keepdims=True)

        #add to dictionary and perform regularization

    #voodoo programmed to bring down the error by randomly adding a transpose␣
↪to W1 and W2 lol
    #i guessed that I had to b/c I had to add regularization and the dimensions␣
↪didn't match unless I transposed
    #i had to add regularization otherwise the error was bad rip

    grads['W1'] = dW.T
    grads['W2'] = dW2.T
    grads['b1'] = db
    grads['b2'] = db2

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

    # ============================================================= #
    # 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 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 np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ========================================================================= #
      # YOUR CODE HERE:
            #         Create a minibatch by sampling batch_size samples␣
↪randomly.
            # ========================================================================= #

      #same code as softmax.py train function
      indices = np.random.choice(np.arange(num_train), batch_size);
      X_batch = X[indices]
      y_batch = y[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␣
↪update
          #         all parameters (i.e., W1, W2, b1, and b2).
            # ========================================================================= #

      #step size is the same as learning rate
      self.params['W1'] -= learning_rate * grads['W1']
```

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

      #can't use += below, see https://stackoverflow.com/questions/47493559/
↪valueerror-non-broadcastable-output-operand-with-shape-3-1-doesnt-match-the/
↪47493938

      self.params['b1'] = self.params['b1'] - (learning_rate * grads['b1'])
      self.params['b2'] = self.params['b2'] - (learning_rate * grads['b2'])

      # ========================================================== #
      # 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 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
        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 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.

    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

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

    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']

    #the same as in loss()

    f = lambda x : x * (x > 0) #relu activation function
    h = f(np.dot(X, W1.T) + b1)
    scores = np.dot(h, W2.T) + b2
    y_pred = np.argmax(scores, axis=1)


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

    return y_pred
```