This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

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

Please print out the notebook entirely when completed.

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

In [215]:
```
import random
import numpy as np
from utils.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))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

In [216]:
```
from nndl.neural_net import TwoLayerNet
```

In [217]:
```
# 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()
```

Compute forward pass scores

In [218]:
```
## Implement the forward pass of the neural network.
## See the loss() method in TwoLayerNet class for the same

# 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:
3.3812311957259755e-08
```

Forward pass loss

In [219]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

```
Loss: 1.071696123862817
Difference between your loss and correct loss:
0.0
```

Backward pass

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

In [220]:
```
from utils.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])))
```

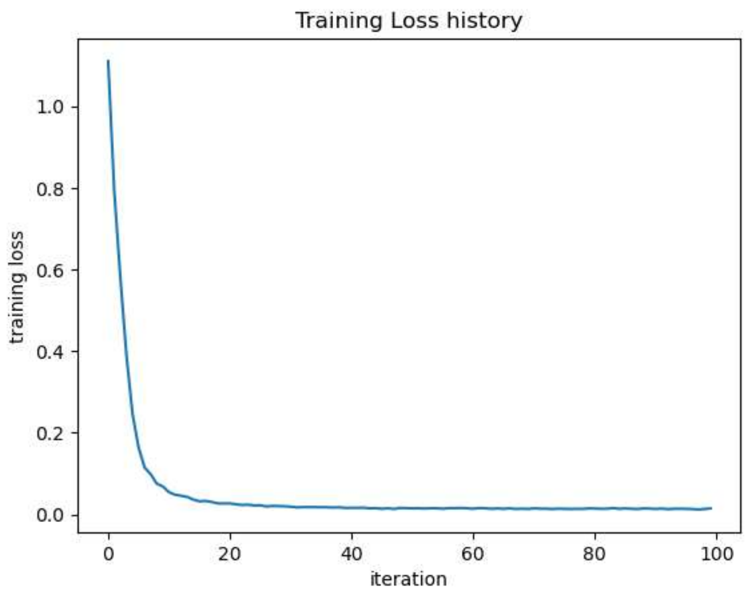
```
W2 max relative error: 2.9632245016399034e-10
b2 max relative error: 1.2482633693659668e-09
W1 max relative error: 1.28328951808708e-09
b1 max relative error: 3.172680285697327e-09
```

Training the network

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

In [221]:
```
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:  0.01449890295297172
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [222]:
```
from utils.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.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '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,)
```

Running SGD

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

In [223]:
```
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 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

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.

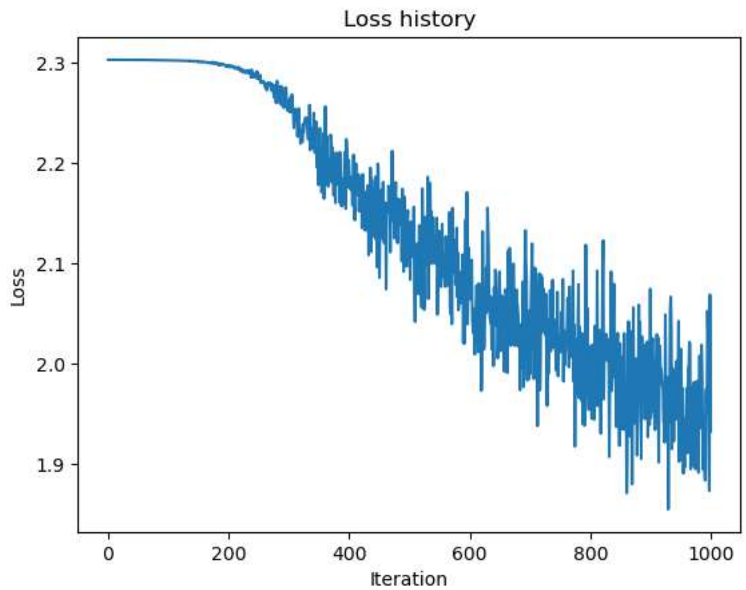
(2) How should you fix the problems you identified in (1)?

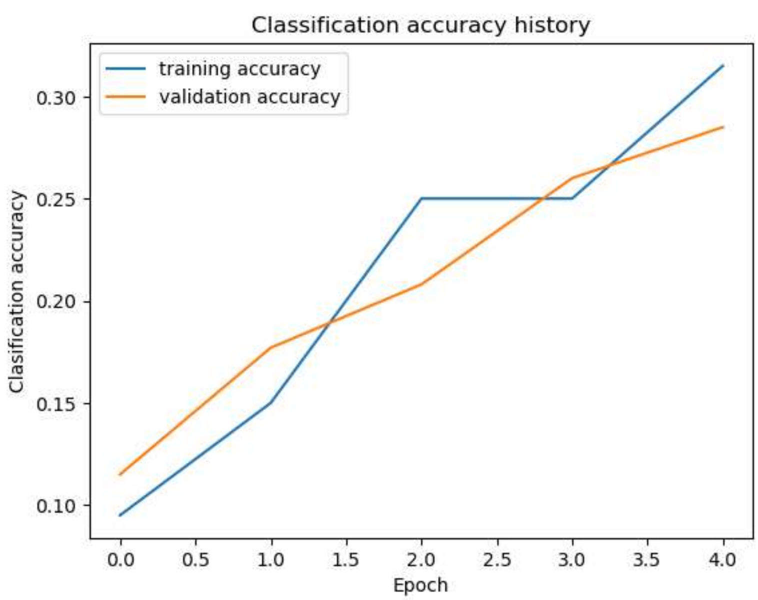
In [224]:
```
stats['train_acc_history']
```

Out[224]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

In [225]:

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

plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

plt.plot(stats['train_acc_history'], label='training accuracy')
plt.plot(stats['val_acc_history'], label='validation accuracy')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()

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





Answers:

(1) From the loss history graph, we see that the loss is stable for the first 200 iterations or so and only decreases after. This means that the learning rate may not be large enough. We also see, from the accuracy history graph, that the accuracies do not converge, which means we may need to increase the number of iterations.

(2) Try larger learning rates and more iterations. Additionally, we can try to change the regularization factor.

Optimize the neural network

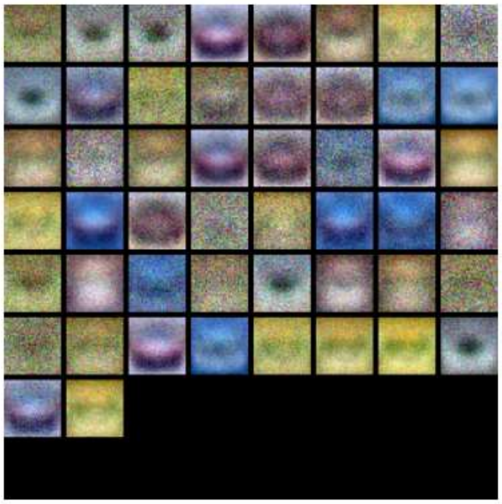
Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

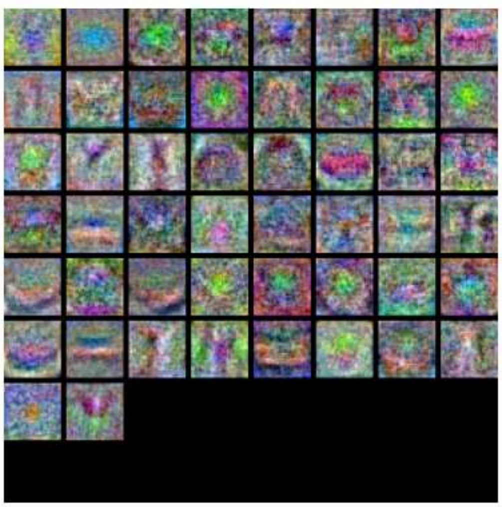
In [226]:

```
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)!
# ================================================================ #
lr = [1e-2, 5e-3, 1e-3, 5e-4]
regs = [0, 0.1, 0.2, 0.3, 0.4, 0.5]

best_accuracy = 0

for l in lr:
    for r in regs:
        nn = TwoLayerNet(input_size, hidden_size, num_classes)

        nn.train(X_train, y_train, X_val, y_val, num_iters=3000, batch_size=200,
            learning_rate=l, learning_rate_decay=0.95,
            reg=r, verbose=False)

        train_acc = (nn.predict(X_train) == y_train).mean()
        val_acc = (nn.predict(X_val) == y_val).mean()

        print("Learning rate:", l, "Regularization factor:", r, "Train accuracy:", train_acc, "Val accuracy:", val_acc)

        if best_accuracy < val_acc:
            best_accuracy = val_acc
            best_net = nn
            best_r = r
            best_l = l

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Best learning rate:', best_l, "Best regularization factor:", best_r)
print('Validation accuracy: ', val_acc)
```

```
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:111: RuntimeWarning: divide by zero encountered in log
  log_loss = -np.log(np.exp(a_y) / sm_sum)
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:110: RuntimeWarning: overflow encountered in exp
  sm_sum = np.sum(np.exp(scores), axis=1)
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:111: RuntimeWarning: overflow encountered in exp
  log_loss = -np.log(np.exp(a_y) / sm_sum)
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:111: RuntimeWarning: invalid value encountered in true_divide
  log_loss = -np.log(np.exp(a_y) / sm_sum)
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:130: RuntimeWarning: overflow encountered in exp
  probs = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
C:\Users\bmang\Desktop\c147\HW3_code\nndl\neural_net.py:130: RuntimeWarning: invalid value encountered in true_divide
  probs = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)

Learning rate: 0.01 Regularization factor: 0 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.01 Regularization factor: 0.1 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.01 Regularization factor: 0.2 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.01 Regularization factor: 0.3 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.01 Regularization factor: 0.4 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.01 Regularization factor: 0.5 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0.1 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0.2 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0.3 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0.4 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.005 Regularization factor: 0.5 Train accuracy: 0.10026530612244898 Val accuracy: 0.087
Learning rate: 0.001 Regularization factor: 0 Train accuracy: 0.5650204081632653 Val accuracy: 0.52
Learning rate: 0.001 Regularization factor: 0.1 Train accuracy: 0.5589387755102041 Val accuracy: 0.506
Learning rate: 0.001 Regularization factor: 0.2 Train accuracy: 0.5640204081632653 Val accuracy: 0.498
Learning rate: 0.001 Regularization factor: 0.3 Train accuracy: 0.5456122448979592 Val accuracy: 0.485
Learning rate: 0.001 Regularization factor: 0.4 Train accuracy: 0.5445102040816326 Val accuracy: 0.51
Learning rate: 0.001 Regularization factor: 0.5 Train accuracy: 0.5499183673469388 Val accuracy: 0.513
Learning rate: 0.0005 Regularization factor: 0 Train accuracy: 0.5249387755102041 Val accuracy: 0.486
Learning rate: 0.0005 Regularization factor: 0.1 Train accuracy: 0.5286326530612245 Val accuracy: 0.498
Learning rate: 0.0005 Regularization factor: 0.2 Train accuracy: 0.5248571428571429 Val accuracy: 0.5
Learning rate: 0.0005 Regularization factor: 0.3 Train accuracy: 0.5228163265306123 Val accuracy: 0.489
Learning rate: 0.0005 Regularization factor: 0.4 Train accuracy: 0.518734693877551 Val accuracy: 0.479
Learning rate: 0.0005 Regularization factor: 0.5 Train accuracy: 0.519061224489796 Val accuracy: 0.499
Best learning rate: 0.001 Best regularization factor: 0
Validation accuracy:  0.52
```

In [227]:
```
from utils.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)
```





Question:

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

Answer:

(1) I can hardly tell the difference between the images for the suboptimal net - the images look like blobs at best and a whole bunch of noise at worst. The weights for the suboptimal net are smoothed out and result in a lot of noise. At least for the best net, different shapes are distinguishable.

Evaluate on test set

In [228]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.491
```

In [1]:
```
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
  D, 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.
    # ================================================================ #

    hl1 = X @ W1.T + b1
    relu = np.maximum(hl1, 0)
    scores = relu @ W2.T + b2
```

```
    # ================================================================ #
    # 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 teh variable loss.  Multiply the regularization
    #   loss by 0.5 (in addition to the factor reg).
    # ================================================================ #

    # scores is num_examples by num_classes
    N = X.shape[0]
    a_y = scores[range(N), y]
    sm_sum = np.sum(np.exp(scores), axis=1)
    log_loss = -np.log(np.exp(a_y) / sm_sum)
    normalized_sm_loss = np.sum(log_loss) / N

    reg_loss = 0.5 * reg * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    loss = normalized_sm_loss + 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.
    # ================================================================ #

    probs = np.exp(scores) / np.sum(np.exp(scores), axis=1, keepdims=True)
    probs[range(N), y] -= 1
    dLda = probs / N

    H = W1.shape[0]
    # after relu layer: upstream derivative is from softmax
    local_grad = np.maximum(W1 @ X.T + b1.reshape(H, 1), 0)
    grads['W2'] = dLda.T @ local_grad.T + reg * W2
    grads['b2'] = np.sum(dLda, axis=0)

    # dLdh is W2.T; derivative is positive only if relu is activated
    upstream = W2.T @ dLda.T
    upstream = (W1 @ X.T > 0) * upstream
    grads['W1'] = upstream @ X + reg * W1
    grads['b1'] = np.sum(upstream, axis=1)

    # ================================================================ #
    # 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.
      # ================================================================ #
      random_indices = np.random.choice(num_train, batch_size)
      X_batch = X[random_indices]
      y_batch = y[random_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).
      # ================================================================ #

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

    hl1 = X @ self.params['W1'].T + self.params['b1']
```

```
        relu = np.maximum(0, hl1)
        scores = relu @ self.params['W2'].T + self.params['b2']
        y_pred = np.argmax(scores, axis=1)

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

        return y_pred
```

In [ ]: