This is the 2-layer neural network workbook for ECE 239AS 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.

```
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))))
```

Toy example

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

```
from nndl.neural_net import TwoLayerNet
```

```python
# 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

```python
## 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:
3.38123121099e-08
```

Forward pass loss

```
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)))
```

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

```
print(loss)
```

```
1.07169612386
```

Backward pass

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

```
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])))
```

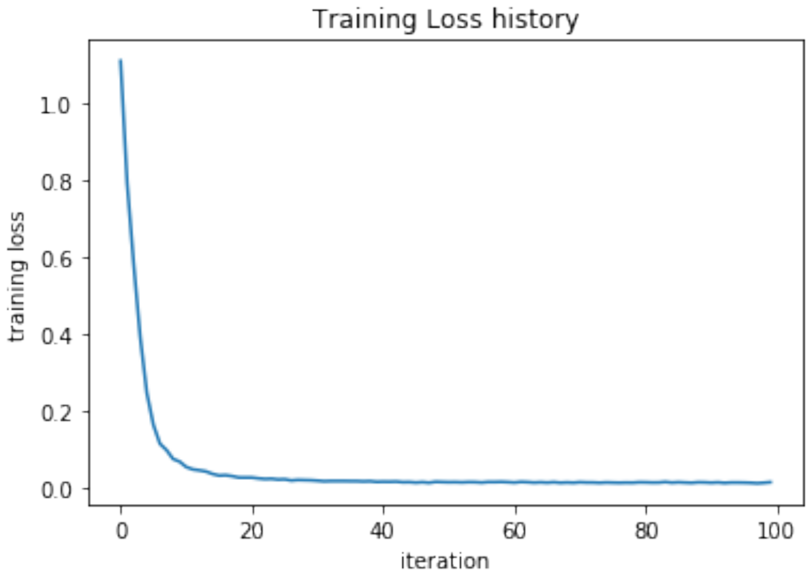
```
b1 max relative error: 3.1726804786908923e-09
W2 max relative error: 2.96322045694799e-10
b2 max relative error: 1.2482669498248223e-09
W1 max relative error: 1.2832845443256344e-09
```

Training the network

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

```
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:  0.0144978645878
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```python
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 = '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%.

```python
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 2.302799087894744
iteration 100 / 1000: loss 2.3020978009578696
iteration 200 / 1000: loss 2.2971055069229687
iteration 300 / 1000: loss 2.25653438333671
iteration 400 / 1000: loss 2.1488124391398284
iteration 500 / 1000: loss 2.07206696122405
iteration 600 / 1000: loss 2.0221237734297546
iteration 700 / 1000: loss 2.0253953947024037
iteration 800 / 1000: loss 1.95275530959424
iteration 900 / 1000: loss 1.965974642337038
Validation accuracy:  0.282
```

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.

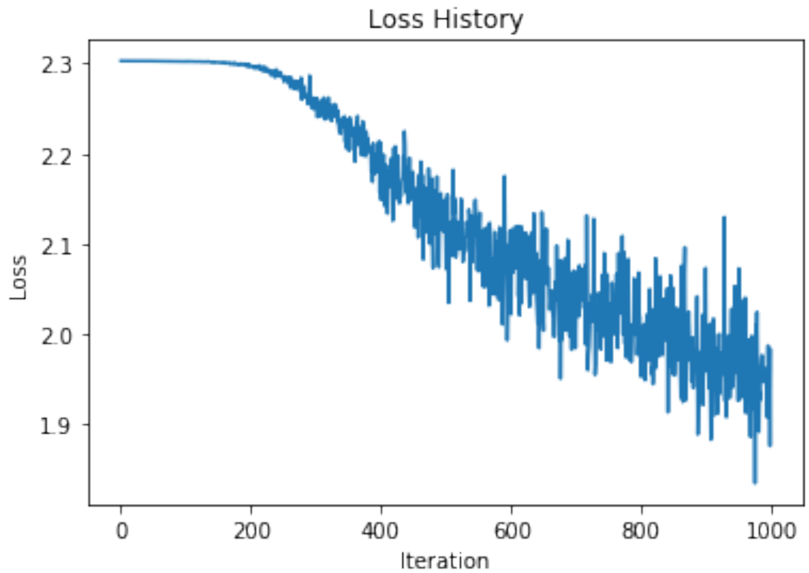
(2) How should you fix the problems you identified in (1)?

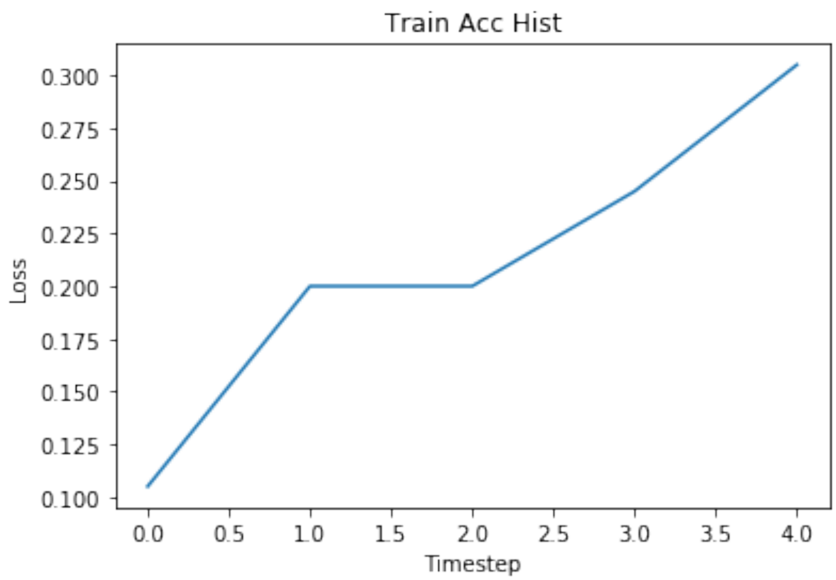
```
stats['train_acc_history']
```

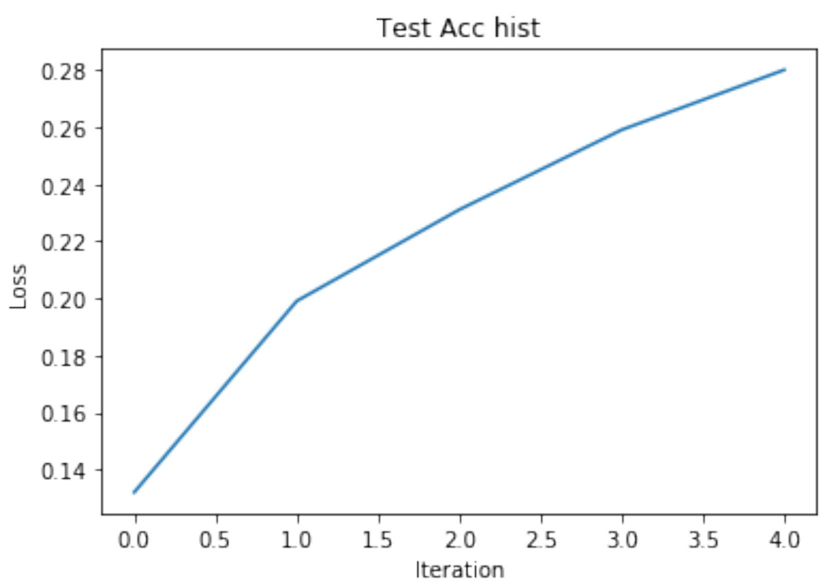
```
[0.105, 0.20000000000000001, 0.20000000000000001, 0.245, 0.30499999999999999]
```

```
# ================================================================ #
# 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
import matplotlib.pyplot as plt
plt.figure()
plt.plot(stats['loss_history'])
plt.title('Loss History')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.figure()
plt.plot(stats['train_acc_history'])
plt.title('Train Acc Hist')
plt.xlabel('Timestep')
plt.ylabel('Loss')
plt.figure()
plt.plot(stats['val_acc_history'])
plt.title('Test Acc hist')
plt.xlabel('Iteration')
plt.ylabel('Loss')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
<matplotlib.text.Text at 0x10fe24978>
```


Loss History
Loss
Iteration
2.3
2.2
2.1
2.0
1.9
0
200
400
600
800
1000


Train Acc Hist
Loss
Timestep
0.300
0.275
0.250
0.225
0.200
0.175
0.150
0.125
0.100
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Test Acc hist
Loss
Iteration
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0.14
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Answers:

(1) Looking at the loss plot, it seems that the network seems to not learn much until 200 iterations in, and then the loss starts descending. The loss is very noisy (probably due to SGD) but overall it seems to be going down. Similarly, the accuracy plots indicate that the accuracy was going up at a relatively consistent rate (i.e. we didn't level off) when we finished iterating. From this, I concluded that the training accuracy wasn't great because we simply didn't give it enough iterations for the loss to converge. This would be my first step in attempting to increase the accuracy. After gauging how many iterations it takes for the loss to roughly converge, I would

(2) My first step in attempting to increase the accuracy would be to increase the number of iterations. After gauging how many iterations it takes for the loss to roughly converge, I would then begin to select my hyperparameters, using performance on a validation dataset. In order to do this, I would probably choose many different settings of hyperparameters that we can change - including batch size, learning rate, decay rate, and regularization strength, and run several different networks to see which setting of hyperparameters result in the best training accuracy.

Optimize the neural network

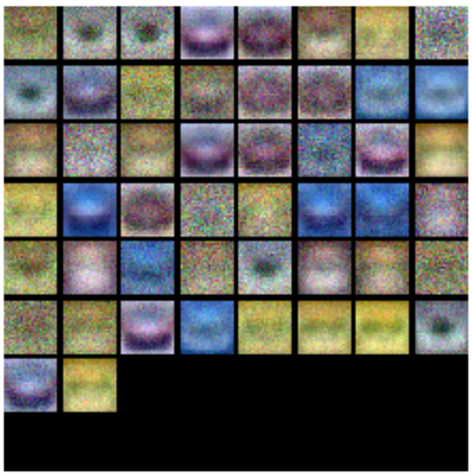
Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

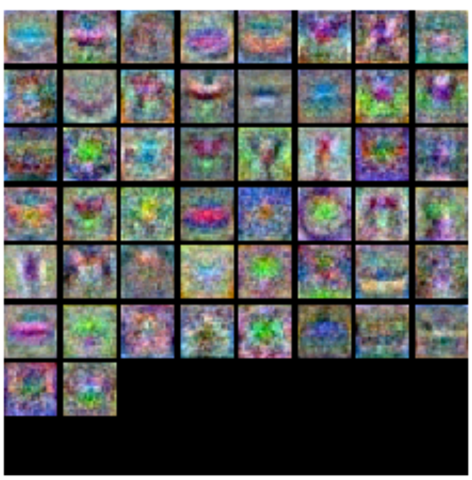
```
best_net = None # store the best model into this

# ================================================================ #
# 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)!
# ================================================================ #


#NOTE: this was the code I used to find the best hyperparameters
# I basically manually listed a bunch of candidate hyperparameters, and then
trained up to 50 networks
# on random hyperparameters. I wrote code to break when we find something over
50%
# Luckily for me, this happened on the first try with hyperparams: num_iters =
10000, batch size = 200, learning_rate = 0.001, decay = 0.85, reg = 0.25
```

```python
# best hyperparams found: num_iters = 10000, batch size = 200, learning_rate = 
0.001, decay = 0.85, reg = 0.25
# as can be seen in the results printed below
batchsizes = [ 50, 100, 200, 500, 1000]
n_iters = 10000
learning_rates = [1e-5, 1e-4, 5e-4, 3e-4, 1e-3, 1e-2, 3e-3, 5e-4]
decays = [0.5, 0.75, 0.9, 0.95, 0.99, 0.85]
reg_strength = [0.1, 0.25, 0.35, 0.45, 0.6, 0.75, 0.9]
# now, train 50 networks with random hyperparameters.
best_val_acc, best_net = 0.0, None
for i in range(50):
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    batchsize = np.random.choice(batchsizes)
    lr = np.random.choice(learning_rates)
    decay = np.random.choice(decays)
    reg = np.random.choice(reg_strength)
    print('training with num_iters = {}, batch size = {}, learning_rate = {}, 
decay = {}, reg = {}'.format(n_iters, batchsize, lr, decay, reg))
    stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=n_iters, batch_size=batchsize,
            learning_rate=lr, learning_rate_decay=decay,
            reg=reg, verbose=False)
    val_acc = (net.predict(X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)
    if val_acc >= 0.5:
        print('best hyperparams found: num_iters = {}, batch size = {}, 
learning_rate = {}, decay = {}, reg = {}'.format(n_iters, batchsize, lr, 
decay, reg))
        best_net = net
        break # found a good enough net
    elif val_acc > best_val_acc:
        best_val_acc = val_acc
        best_net = net
        print('found acc {}'.format(best_val_acc))

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

```
training with num_iters = 10000, batch size = 200, learning_rate = 0.001, 
decay = 0.85, reg = 0.25
Validation accuracy:  0.502
best hyperparams found: num_iters = 10000, batch size = 200, learning_rate = 
0.001, decay = 0.85, reg = 0.25
```

```python
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)
```





Question:

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

Answer:

(1) The images that come from the suboptimal net weights appear quite noisy, with very subtle, hard to make out templates for some of the images they may be capable of detecting. on the other hand, the best net I arrived out has much more robust shapes and colors in their weight visualization. FOr example, some of the images coming from the weight visualization appear like a well-defined red car. In the best net weights, we see a much clearer shape in terms of objects and colors, while the suboptimal net, these shapes and colors were much less well defined and a lot noisier. Also, in the suboptimal net, a lot of the templates seem to be faint outlines of a red car, which indicates that the weights did not learn very much about the other types of images, while each weight visualization in the optimal net is quite different (and more well defined).

Evaluate on test set

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

```
Test accuracy:  0.506
```

Fully connected networks

In the previous notebook, you implemented a simple two-layer neural network class. However, this class is not modular. If you wanted to change the number of layers, you would need to write a new loss and gradient function. If you wanted to optimize the network with different optimizers, you'd need to write new training functions. If you wanted to incorporate regularizations, you'd have to modify the loss and gradient function.

Instead of having to modify functions each time, for the rest of the class, we'll work in a more modular framework where we define forward and backward layers that calculate losses and gradients respectively. Since the forward and backward layers share intermediate values that are useful for calculating both the loss and the gradient, we'll also have these function return "caches" which store useful intermediate values.

The goal is that through this modular design, we can build different sized neural networks for various applications.

In this HW #3, we'll define the basic architecture, and in HW #4, we'll build on this framework to implement different optimizers and regularizations (like BatchNorm and Dropout).

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

Modular layers

This notebook will build modular layers in the following manner. First, there will be a forward pass for a given layer with inputs (`x`) and return the output of that layer (`out`) as well as cached variables (`cache`) that will be used to calculate the gradient in the backward pass.

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,
eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-
ipython
%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))))
```

```python
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_val: (1000, 3, 32, 32)
y_test: (1000,)
y_train: (49000,)
X_test: (1000, 3, 32, 32)
y_val: (1000,)
X_train: (49000, 3, 32, 32)
```

Linear layers

In this section, we'll implement the forward and backward pass for the linear layers.

The linear layer forward pass is the function `affine_forward` in `nndl/layers.py` and the backward pass is `affine_backward`.

After you have implemented these, test your implementation by running the cell below.

Affine layer forward pass

Implement `affine_forward` and then test your code by running the following cell.

```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),
output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing affine_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing affine_forward function:
difference: 9.7698500479884e-10
```

Affine layer backward pass

Implement `affine_backward` and then test your code by running the following cell.

```
# Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0],
x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0],
w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0],
b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around 1e-10
print('Testing affine_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_backward function:
dx error: 1.7792258869473942e-10
dw error: 1.7504796033291886e-10
db error: 1.7519769156660496e-11
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

Implement the `relu_forward` function in `nndl/layers.py` and then test your code by running the following cell.

```python
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,         ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,
]])

# Compare your output with ours. The error should be around 1e-8
print('Testing relu_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing relu_forward function:
difference: 4.999999798022158e-08
```

ReLU backward pass

Implement the `relu_backward` function in `nndl/layers.py` and then test your code by running the following cell.

```python
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print('Testing relu_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing relu_backward function:
dx error: 3.2756113236027615e-12
```

Combining the affine and ReLU layers

Often times, an affine layer will be followed by a ReLU layer. So let's make one that puts them together. Layers that are combined are stored in `nndl/layer_utils.py`.

Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nndl/layer_utils.py`. Take a look at them to make sure you understand what's going on. Then run the following cell to ensure its implemented correctly.

```
from nndl.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)
[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)
[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)
[0], b, dout)

print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error: 2.261033893905029e-09
dw error: 6.051708743538486e-09
db error: 7.55627367037208e-11
```

Softmax and SVM losses

You've already implemented these, so we have written these in `layers.py`. The following code will ensure they are working correctly.

```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x,
verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,
verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing svm_loss:
loss: 8.998306995796787
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.3024162325284747
dx error: 7.635214473842447e-09
```

Implementation of a two-layer NN

In `nndl/fc_net.py`, implement the class `TwoLayerNet` which uses the layers you made here. When you have finished, the following cell will test your implementation.

```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dims=H, num_classes=C,
weight_scale=std)

print('Testing initialization ... ')
```

```
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434,
15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128,
15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822,
15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'
print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'
model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = {}'.format(reg))
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('{} relative error: {}'.format(name, rel_error(grad_num,
grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 2.131611955458401e-08
W2 relative error: 3.310270199776237e-10
b1 relative error: 8.368196732475888e-09
b2 relative error: 2.530774050159566e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279153413239097e-07
W2 relative error: 1.367837124985045e-07
b1 relative error: 1.5646802033932055e-08
b2 relative error: 9.089614638133234e-10
```

Solver

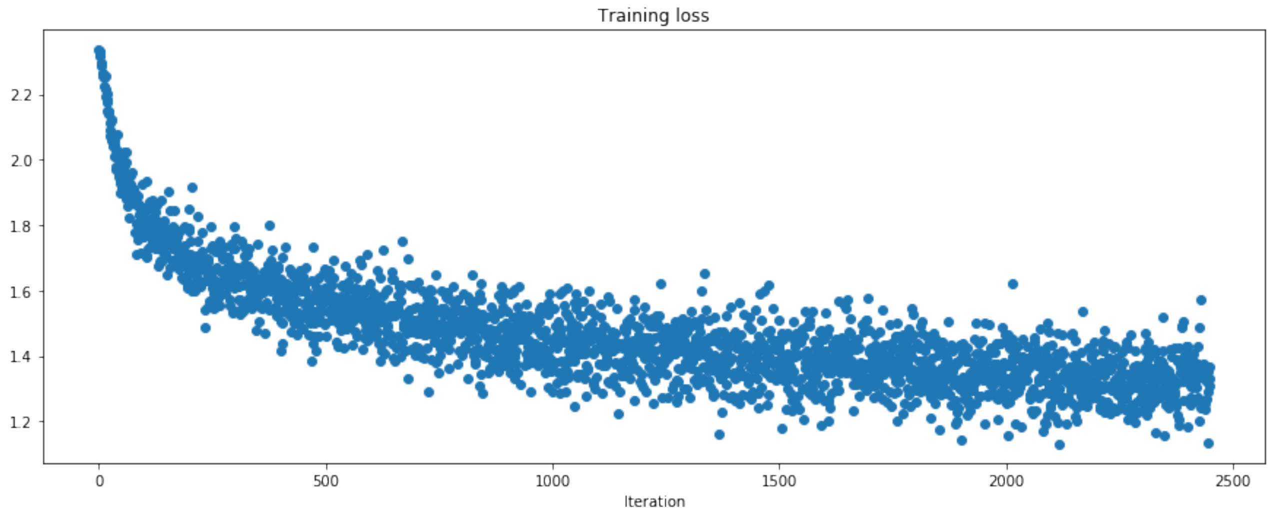
We will now use the cs231n Solver class to train these networks. Familiarize yourself with the API in `cs231n/solver.py`. After you have done so, declare an instance of a TwoLayerNet with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least 50%.

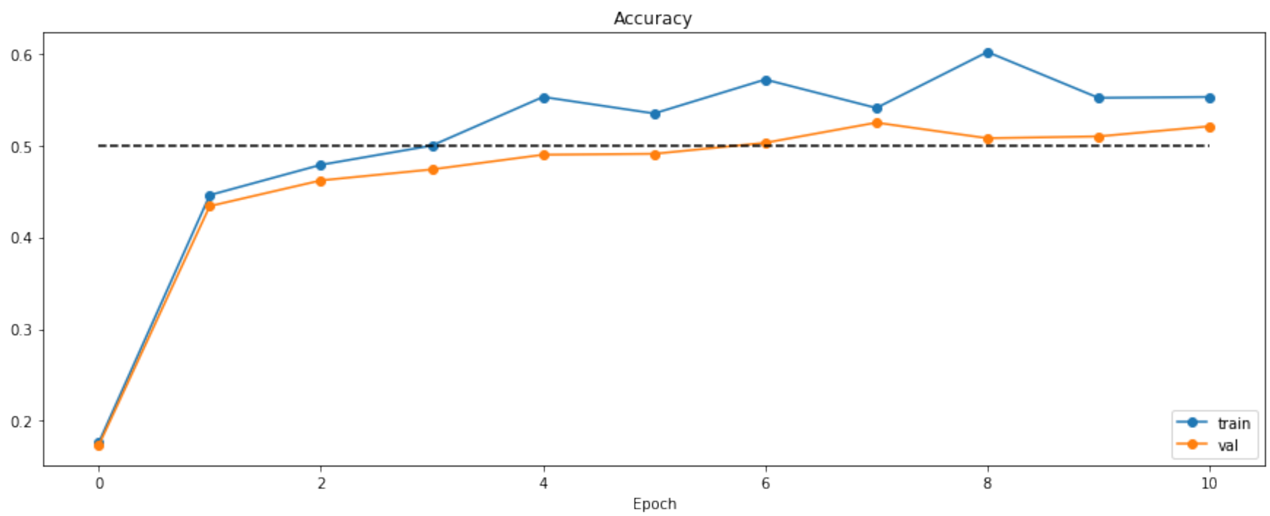
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 40%.  We won't have you optimize this further
#   since you did it in the previous notebook.
# ================================================================ #

model = TwoLayerNet(reg = 0.25)
solver = Solver(model, data = data,
               optim_config = {
                   'learning_rate': 0.001,
               }, lr_decay = 0.85, num_epochs = 10, batch_size = 200,
print_every = 50)
solver.train()

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

```
(Iteration 1 / 2450) loss: 2.337337
(Epoch 0 / 10) train acc: 0.176000; val_acc: 0.173000
(Iteration 51 / 2450) loss: 1.910711
(Iteration 101 / 2450) loss: 1.802551
(Iteration 151 / 2450) loss: 1.746674
(Iteration 201 / 2450) loss: 1.782862
(Epoch 1 / 10) train acc: 0.446000; val_acc: 0.434000
(Iteration 251 / 2450) loss: 1.606931
(Iteration 301 / 2450) loss: 1.609461
(Iteration 351 / 2450) loss: 1.620807
(Iteration 401 / 2450) loss: 1.636927
(Iteration 451 / 2450) loss: 1.644888
(Epoch 2 / 10) train acc: 0.479000; val_acc: 0.462000
(Iteration 501 / 2450) loss: 1.590001
(Iteration 551 / 2450) loss: 1.539093
(Iteration 601 / 2450) loss: 1.500296
(Iteration 651 / 2450) loss: 1.633473
(Iteration 701 / 2450) loss: 1.552428
(Epoch 3 / 10) train acc: 0.500000; val_acc: 0.474000
(Iteration 751 / 2450) loss: 1.498844
(Iteration 801 / 2450) loss: 1.485141
(Iteration 851 / 2450) loss: 1.401013
(Iteration 901 / 2450) loss: 1.536564
(Iteration 951 / 2450) loss: 1.449009
(Epoch 4 / 10) train acc: 0.553000; val_acc: 0.490000
(Iteration 1001 / 2450) loss: 1.289351
(Iteration 1051 / 2450) loss: 1.448503
(Iteration 1101 / 2450) loss: 1.463511
(Iteration 1151 / 2450) loss: 1.381720
(Iteration 1201 / 2450) loss: 1.302922
(Epoch 5 / 10) train acc: 0.535000; val_acc: 0.491000
(Iteration 1251 / 2450) loss: 1.344926
(Iteration 1301 / 2450) loss: 1.369209
(Iteration 1351 / 2450) loss: 1.359285
(Iteration 1401 / 2450) loss: 1.349837
(Iteration 1451 / 2450) loss: 1.398131
(Epoch 6 / 10) train acc: 0.572000; val_acc: 0.503000
(Iteration 1501 / 2450) loss: 1.425162
(Iteration 1551 / 2450) loss: 1.299397
(Iteration 1601 / 2450) loss: 1.266481
(Iteration 1651 / 2450) loss: 1.573615
(Iteration 1701 / 2450) loss: 1.410486
(Epoch 7 / 10) train acc: 0.541000; val_acc: 0.525000
(Iteration 1751 / 2450) loss: 1.418999
(Iteration 1801 / 2450) loss: 1.258174
(Iteration 1851 / 2450) loss: 1.329931
```

```
(Iteration 1901 / 2450) loss: 1.395133
(Iteration 1951 / 2450) loss: 1.328131
(Epoch 8 / 10) train acc: 0.602000; val_acc: 0.508000
(Iteration 2001 / 2450) loss: 1.254937
(Iteration 2051 / 2450) loss: 1.391966
(Iteration 2101 / 2450) loss: 1.434608
(Iteration 2151 / 2450) loss: 1.311771
(Iteration 2201 / 2450) loss: 1.374558
(Epoch 9 / 10) train acc: 0.552000; val_acc: 0.510000
(Iteration 2251 / 2450) loss: 1.319331
(Iteration 2301 / 2450) loss: 1.273286
(Iteration 2351 / 2450) loss: 1.328836
(Iteration 2401 / 2450) loss: 1.327306
(Epoch 10 / 10) train acc: 0.553000; val_acc: 0.521000
```

```python
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `nndl/fc_net.py`.

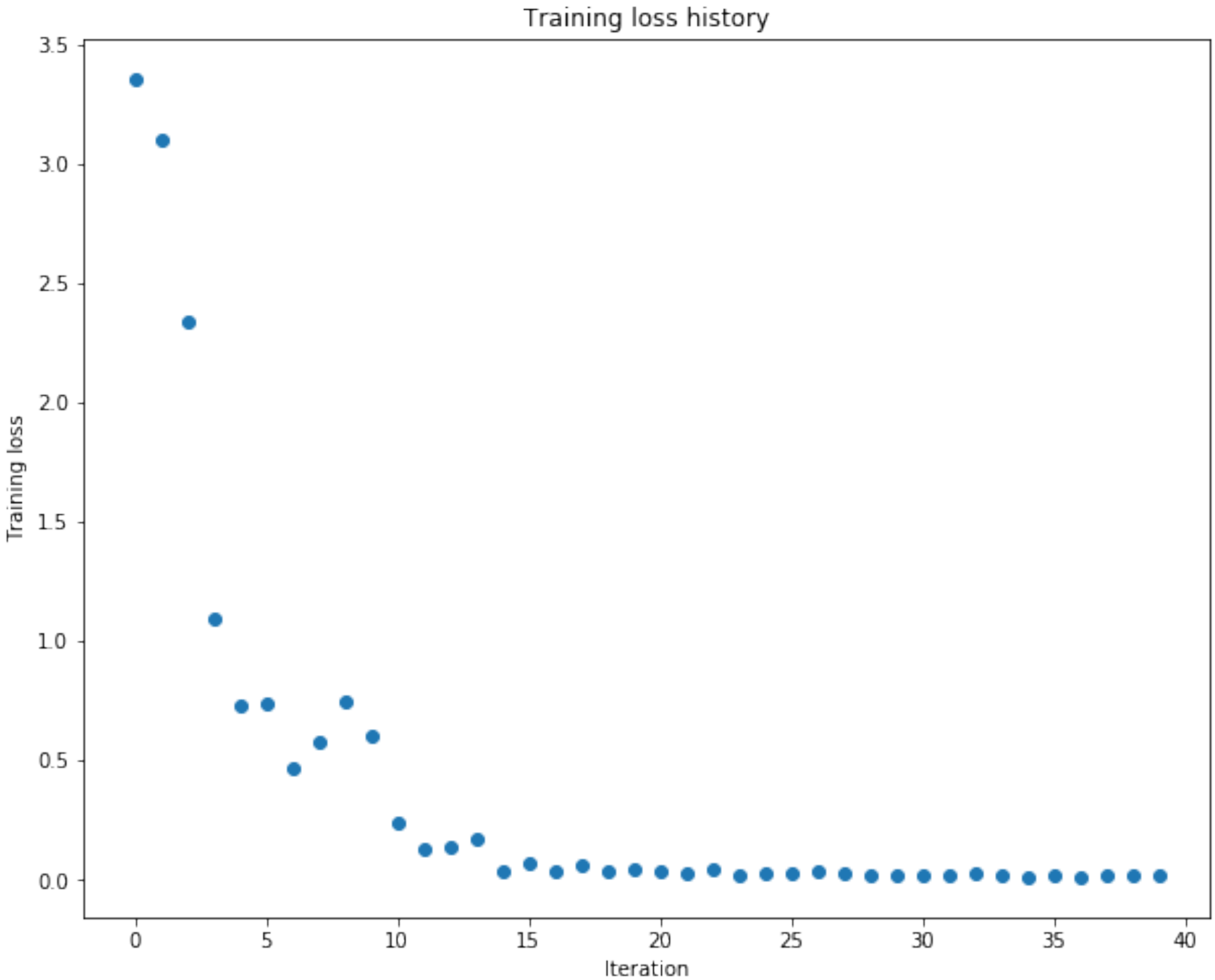
Implement the initialization, the forward pass, and the backward pass. There will be lines for batchnorm and dropout layers and caches; ignore these all for now. That'll be in assignment #4.

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = {}'.format(reg))
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: {}'.format(loss))

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,
h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num,
grads[name])))
```

```
Running check with reg = 0
Initial loss: 2.3227094181176984
W1 relative error: 9.473423981398794e-06
W2 relative error: 5.335962923177645e-09
b1 relative error: 6.073247014072795e-09
b2 relative error: 1.328227452893485e-10
Running check with reg = 3.14
Initial loss: 4.3230386136289765
W1 relative error: 3.800613468132099e-08
W2 relative error: 3.117433723741423e-08
b1 relative error: 6.036366525813558e-09
b2 relative error: 1.750006524595143e-10
```

```
# Use the three layer neural network to overfit a small dataset.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}



#### !!!!!!
# Play around with the weight_scale and learning_rate so that you can overfit
a small dataset.
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-4
# i just changed the learning rate to .001.
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 0.001,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 3.355742
(Epoch 0 / 20) train acc: 0.380000; val_acc: 0.114000
(Epoch 1 / 20) train acc: 0.440000; val_acc: 0.153000
(Epoch 2 / 20) train acc: 0.720000; val_acc: 0.120000
(Epoch 3 / 20) train acc: 0.800000; val_acc: 0.146000
(Epoch 4 / 20) train acc: 0.880000; val_acc: 0.141000
(Epoch 5 / 20) train acc: 0.960000; val_acc: 0.163000
(Iteration 11 / 40) loss: 0.237550
(Epoch 6 / 20) train acc: 0.980000; val_acc: 0.152000
(Epoch 7 / 20) train acc: 1.000000; val_acc: 0.150000
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.155000
(Iteration 21 / 40) loss: 0.034686
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.152000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.153000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.151000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.158000
(Iteration 31 / 40) loss: 0.018596
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.159000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.159000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.160000
```



Training loss history
Training loss
Iteration
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35
40

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 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 (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.
# ================================================================ #
z1 = X.dot(W1.T) + b1
h1 = z1 * (z1 > 0)
z2 = h1.dot(W2.T) + b2
scores = z2

# ================================================================ #
# 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

# first, normalize the scores
normscores = scores - np.max(scores, axis = 1, keepdims = True)
# now calculate the softmax function
score_probs = np.exp(normscores)
score_probs/=np.sum(score_probs, axis = 1, keepdims = True)
# now, pick out the correct ones, sum, and norm
d_loss = -np.sum(np.log(score_probs[np.arange(scores.shape[0]),
y].clip(min=np.finfo(float).eps))) / scores.shape[0]
# penalize by frobenius norm
r_loss = .5 * reg * (np.sum(W1 **2) + np.sum(W2 ** 2))
#loss+= 0.5 * reg * np.sum(W1**2) + 0.5 * reg * np.sum(W2 ** 2)
loss = d_loss + r_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.
# ================================================================ #

# calculate the grad with respect to softmax
p = score_probs.copy()
```

```
# to account for the case where w_j = w_{y_i} (i.e. our class corresponds to
the correct class label)
# in the unvectorized version we multiplied by -x[i], so here we -1
p[range(X.shape[0]),y]-=1
p/=X.shape[0]
# now back to the bias
# the chain rule means just times it by 1
dldb2 = np.sum(p, axis = 0)
# now back to the second layer weights
# since we computed Wh1 where h1 was the input into this layer, derivative is
h1, and add derivative of regularization func
dldw2 = p.T.dot(h1) + reg * W2
# calculate the gradient that we send back
# basically this is the gradient of the inputs into this layer, h1
# since we did Wh1 the grad is just W, times p  for the chain rule
dLdh1 = p.dot(W2) # this is the "upstream gradient" for the first layer, where
as p was the upstream for second layer
# now back into the relu
dldz = dLdh1 * (z1 > 0)
# now back into the first layer bias
dldb1 = np.sum(dldz, axis = 0)
dldw1 = dldz.T.dot(X) + reg * W1
# now back into the first layer weights

# assign grads
grads['b2'] = dldb2
grads['W2'] = dldw2
grads['b1'] = dldb1
grads['W1'] = dldw1
# ================================================================ #
# 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.
  # ================================================================ #
  indices = np.random.choice(X.shape[0], 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).
  # ================================================================ #
  self.params['b1']+=-learning_rate*grads['b1']
```

```
    self.params['W1']+=-learning_rate*grads['W1']
    self.params['b2']+=-learning_rate*grads['b2']
    self.params['W2']+=-learning_rate*grads['W2']

    # ================================================================ #
    # 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.
# ================================================================ #
# get the scores
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
z1 = X.dot(W1.T) + b1
h1 = z1 * (z1 > 0)
z2 = h1.dot(W2.T) + b2
scores = z2
y_pred = np.argmax(scores, axis = 1)
```

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

return y_pred
```

import numpy as np import pdb

""" 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.
"""

def affine_forward(x, w, b): """ Computes the forward pass for an affine (fully-connected) layer.

The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N examples, where each example x[i] has shape (d_1, ..., d_k). We will reshape each input into a vector of dimension D = d_1 * ... * d_k, and then transform it to an output vector of dimension M.

Inputs:

- x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
- w: A numpy array of weights, of shape (D, M)
- b: A numpy array of biases, of shape (M,)

Returns a tuple of:

- out: output, of shape (N, M)
- cache: (x, w, b)
  """

==

YOUR CODE HERE:

Calculate the output of the forward pass. Notice the dimensions

of w are D x M, which is the transpose of what we did in earlier

assignments.

==

---

print('minibatch: {}'.format(x.shape))

prod = np.prod(x.shape[1:]) out = x.reshape((x.shape[0], prod)).dot(w) + b

==

END YOUR CODE HERE

==

---

cache = (x, w, b) return out, cache

def affine_backward(dout, cache): """ Computes the backward pass for an affine layer.

Inputs:

- dout: Upstream derivative, of shape (N, M)

- cache: Tuple of:

    - x: Input data, of shape (N, d_1, ... d_k)
    - w: Weights, of shape (D, M)

Returns a tuple of:

- dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
- dw: Gradient with respect to w, of shape (D, M)
- db: Gradient with respect to b, of shape (M,)
  """
   x, w, b = cache
   dx, dw, db = None, None, None

==

YOUR CODE HERE:

Calculate the gradients for the backward pass.

===

prod = np.prod(x.shape[1:]) db = np.sum(dout, axis = 0) dw = x.reshape((x.shape[0], prod)).T.dot(dout) dx = dout.dot(w.T).reshape(x.shape)

===

END YOUR CODE HERE

===

return dx, dw, db

def relu_forward(x): """ Computes the forward pass for a layer of rectified linear units (ReLUs).

Input:

- x: Inputs, of any shape

Returns a tuple of:

- out: Output, of the same shape as x
- cache: x
"""

===

YOUR CODE HERE:

Implement the ReLU forward pass.

==

out = x * (x > 0)

==

END YOUR CODE HERE

==

cache = x return out, cache

def relu_backward(dout, cache): """ Computes the backward pass for a layer of rectified linear units (ReLUs).

Input:

- dout: Upstream derivatives, of any shape
- cache: Input x, of same shape as dout

Returns:

- dx: Gradient with respect to x
  """
    x = cache

==

YOUR CODE HERE:

Implement the ReLU backward pass

===

the relu backwards pass is like a gate

so if x was > 0 there, the derivative is 1 else 0

dx = dout * (x > 0)

===

END YOUR CODE HERE

===

return dx

def svm_loss(x, y): """ Computes the loss and gradient using for multiclass SVM classification.

Inputs:

- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
  for the ith input.
- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
  0 <= y[i] < C

Returns a tuple of:

- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x
  """
    N = x.shape[0]
    correct_class_scores = x[np.arange(N), y]

```
margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
margins[np.arange(N), y] = 0
loss = np.sum(margins) / N
num_pos = np.sum(margins > 0, axis=1)
dx = np.zeros_like(x)
dx[margins > 0] = 1
dx[np.arange(N), y] -= num_pos
dx /= N
return loss, dx
```

def softmax_loss(x, y): """ Computes the loss and gradient for softmax classification.

Inputs:

- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
  for the ith input.
- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
  0 <= y[i] < C

Returns a tuple of:

- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x
  """

probs = np.exp(x - np.max(x, axis=1, keepdims=True)) probs /= np.sum(probs, axis=1, keepdims=True) N = x.shape[0] loss = -np.sum(np.log(probs[np.arange(N), y])) / N dx = probs.copy() dx[np.arange(N), y] -= 1 dx /= N return loss, dx import numpy as np

from .layers import * from .layer_utils 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 with ReLU nonlinearity and softmax loss that uses a modular layer design. We assume an input dimension of D, a hidden dimension of H, and perform classification over C classes.

The architecure should be affine - relu - affine - softmax.

Note that this class does not implement gradient descent; instead, it will interact with a separate Solver object that is responsible for running optimization.

The learnable parameters of the model are stored in the dictionary self.params that maps parameter names to numpy arrays. """

def **init**(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,

```
            dropout=0, weight_scale=1e-3, reg=0.0):
"""
Initialize a new network.

Inputs:
- input_dim: An integer giving the size of the input
- hidden_dims: An integer giving the size of the hidden layer
- num_classes: An integer giving the number of classes to classify
- dropout: Scalar between 0 and 1 giving dropout strength.
- weight_scale: Scalar giving the standard deviation for random
  initialization of the weights.
- reg: Scalar giving L2 regularization strength.
"""
self.params = {}
self.reg = reg

# ================================================================ #
# YOUR CODE HERE:
#   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
#   self.params['W2'], self.params['b1'] and self.params['b2']. The
#   biases are initialized to zero and the weights are initialized
#   so that each parameter has mean 0 and standard deviation weight_scale.
#   The dimensions of W1 should be (input_dim, hidden_dim) and the
#   dimensions of W2 should be (hidden_dims, num_classes)
# ================================================================ #

self.params['W1'] = np.random.randn(input_dim, hidden_dims) * weight_scale
self.params['b1'] = np.zeros(hidden_dims)
self.params['W2'] = np.random.randn(hidden_dims, num_classes) * weight_scale
self.params['b2'] = np.zeros(num_classes)

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

def loss(self, X, y=None):

```
"""
Compute loss and gradient for a minibatch of data.

Inputs:
- X: Array of input data of shape (N, d_1, ..., d_k)
- y: Array of labels, of shape (N,). y[i] gives the label for X[i].

Returns:
```

```
If y is None, then run a test-time forward pass of the model and return:
- scores: Array of shape (N, C) giving classification scores, where
  scores[i, c] is the classification score for X[i] and class c.

If y is not None, then run a training-time forward and backward pass and
return a tuple of:
- loss: Scalar value giving the loss
- grads: Dictionary with the same keys as self.params, mapping parameter
  names to gradients of the loss with respect to those parameters.
"""
scores = None

# ================================================================== #
# YOUR CODE HERE:
#   Implement the forward pass of the two-layer neural network. Store
#   the class scores as the variable 'scores'.  Be sure to use the layers
#   you prior implemented.
# ================================================================== #

out, cache = affine_relu_forward(X, self.params['W1'], self.params['b1'])
scores, cache_2 = affine_forward(out, self.params['W2'], self.params['b2'])
# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #

# If y is None then we are in test mode so just return scores
if y is None:
  return scores

loss, grads = 0, {}
# ================================================================== #
# YOUR CODE HERE:
#   Implement the backward pass of the two-layer neural net.  Store
#   the loss as the variable 'loss' and store the gradients in the
#   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
#   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
#   i.e., grads[k] holds the gradient for self.params[k].
#
#   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
#   for each W.  Be sure to include the 0.5 multiplying factor to
#   match our implementation.
#
#   And be sure to use the layers you prior implemented.
# ================================================================== #

data_loss, data_loss_grad = softmax_loss(scores, y)
```

```
r_loss = .5 * self.reg * (np.sum(self.params['W1'] **2) +
np.sum(self.params['W2'] ** 2))
loss = data_loss + r_loss
# now backwards through the network
# so affine backwards needs the data_loss_grad and the original input and
weights and bias into the last layer.
# that is cache 2 from above
dx_into_next_layer, dw_2, db_2 = affine_backward(data_loss_grad, cache_2)
grads['W2'] = dw_2 + self.reg * self.params['W2'] # add the regularization
derivative
grads['b2'] = db_2

# now we want to use affine_relu_backwards to do everything for us
# its incoming gradient is the dx_into_next_layer
# we need to assemble a tuple (fc_cache, relu_cache) where fc_cache was the
inputs into this fc layer
# and relu_cache was the inputs into relu
# luckily this is just the cache from affine_relu_forward
dx, dw_1, db_1 = affine_relu_backward(dx_into_next_layer, cache)
grads['W1'] = dw_1 + self.reg * self.params['W1']
grads['b1'] = db_1

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

return loss, grads
```

class FullyConnectedNet(object): """ A fully-connected neural network with an arbitrary number of hidden layers, ReLU nonlinearities, and a softmax loss function. This will also implement dropout and batch normalization as options. For a network with L layers, the architecture will be

{affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

where batch normalization and dropout are optional, and the {...} block is repeated L - 1 times.

Similar to the TwoLayerNet above, learnable parameters are stored in the self.params dictionary and will be learned using the Solver class. """

def **init**(self, hidden_dims, input_dim=3*32*32, num_classes=10,

```
           dropout=0, use_batchnorm=False, reg=0.0,
           weight_scale=1e-2, dtype=np.float32, seed=None):
"""
Initialize a new FullyConnectedNet.

Inputs:
```

```
- hidden_dims: A list of integers giving the size of each hidden layer.
- input_dim: An integer giving the size of the input.
- num_classes: An integer giving the number of classes to classify.
- dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
  the network should not use dropout at all.
- use_batchnorm: Whether or not the network should use batch normalization.
- reg: Scalar giving L2 regularization strength.
- weight_scale: Scalar giving the standard deviation for random
  initialization of the weights.
- dtype: A numpy datatype object; all computations will be performed using
  this datatype. float32 is faster but less accurate, so you should use
  float64 for numeric gradient checking.
- seed: If not None, then pass this random seed to the dropout layers. This
  will make the dropout layers deteriminstic so we can gradient check the
  model.
"""
self.use_batchnorm = use_batchnorm
self.use_dropout = dropout > 0
self.reg = reg
self.num_layers = 1 + len(hidden_dims)
self.dtype = dtype
self.params = {}

# ================================================================ #
# YOUR CODE HERE:
#   Initialize all parameters of the network in the self.params dictionary.
#   The weights and biases of layer 1 are W1 and b1; and in general the
#   weights and biases of layer i are Wi and bi. The
#   biases are initialized to zero and the weights are initialized
#   so that each parameter has mean 0 and standard deviation weight_scale.
# ================================================================ #

for idx, val in enumerate(hidden_dims):
  self.params['W' + str(idx+1)] = np.random.randn(input_dim if idx == 0 else
hidden_dims[idx-1], val if idx != len(hidden_dims) - 1 else num_classes) *
weight_scale
  self.params['b' + str(idx+1)] = np.zeros(val if idx != len(hidden_dims) - 1
else num_classes)

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

# When using dropout we need to pass a dropout_param dictionary to each
# dropout layer so that the layer knows the dropout probability and the mode
# (train / test). You can pass the same dropout_param to each dropout layer.
```

```
self.dropout_param = {}
if self.use_dropout:
  self.dropout_param = {'mode': 'train', 'p': dropout}
  if seed is not None:
    self.dropout_param['seed'] = seed

# With batch normalization we need to keep track of running means and
# variances, so we need to pass a special bn_param object to each batch
# normalization layer. You should pass self.bn_params[0] to the forward pass
# of the first batch normalization layer, self.bn_params[1] to the forward
# pass of the second batch normalization layer, etc.
self.bn_params = []
if self.use_batchnorm:
  self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]

# Cast all parameters to the correct datatype
for k, v in self.params.items():
  self.params[k] = v.astype(dtype)
```

def loss(self, X, y=None):

```
"""
Compute loss and gradient for the fully-connected net.

Input / output: Same as TwoLayerNet above.
"""
X = X.astype(self.dtype)
mode = 'test' if y is None else 'train'

# Set train/test mode for batchnorm params and dropout param since they
# behave differently during training and testing.
if self.dropout_param is not None:
  self.dropout_param['mode'] = mode
if self.use_batchnorm:
  for bn_param in self.bn_params:
    bn_param[mode] = mode

scores = None

# ================================================================= #
# YOUR CODE HERE:
#   Implement the forward pass of the FC net and store the output
#   scores as the variable "scores".
# ================================================================= #
layer_caches = []
input = X
for i in range(1, self.num_layers):
  w, b = self.params['W' + str(i)], self.params['b' + str(i)]
  out, cache = affine_relu_forward(input, w, b) if i != self.num_layers - 1
else affine_forward(out, w, b)
  input = out # input into the next layer is the out of this layer
  layer_caches.append(cache) # store each layer inputs basically
scores = out
```

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

# If test mode return early
if mode == 'test':
  return scores

loss, grads = 0.0, {}
# =================================================================== #
# YOUR CODE HERE:
#   Implement the backwards pass of the FC net and store the gradients
#   in the grads dict, so that grads[k] is the gradient of self.params[k]
#   Be sure your L2 regularization includes a 0.5 factor.
# =================================================================== #

data_loss, data_loss_grad = softmax_loss(scores, y)
r_loss = .5 * self.reg * np.sum([np.sum(self.params['W' + str(i)] **2) for i
in range(1, self.num_layers)])
loss = data_loss + r_loss
# now backprop
incoming_grad = data_loss_grad # the incoming grad into the last layer is the
loss gradient; this will be updated everytime with the newly computed gradient
for i in range(self.num_layers-1, 0, -1):
  w, b = self.params['W' + str(i)], self.params['b' + str(i)]
  # affine_relu_backwards on everything besides the very last layer.
  dx, dw, db = affine_relu_backward(incoming_grad, layer_caches[i-1]) if i !=
self.num_layers - 1 else affine_backward(incoming_grad, layer_caches[i-1])
  grads['W' + str(i)], grads['b' + str(i)] = dw + self.reg * w, db
  incoming_grad = dx # assign the gradient signal incoming into the next layer

# =================================================================== #
# END YOUR CODE HERE
# =================================================================== #
return loss, grads
```