

ECE C147, Winter 2020  
Homework 3  
Submitted by Guy Ohayon

**1 2-layer neural network**

## 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 jupyter 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.

```
In [155]: 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))))
```

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

```
In [156]: from nndl.neural_net import TwoLayerNet
```

```
In [157]: # 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 [158]: ## 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:

3.38123120266e-08

## Forward pass loss

```
In [159]: 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: 1.07169612386

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 [160]: 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: 3.172680092703762e-09
W2 max relative error: 3.425470383805365e-10
W1 max relative error: 1.2832784652838671e-09
b2 max relative error: 1.8391748601536041e-10
```

## Training the network

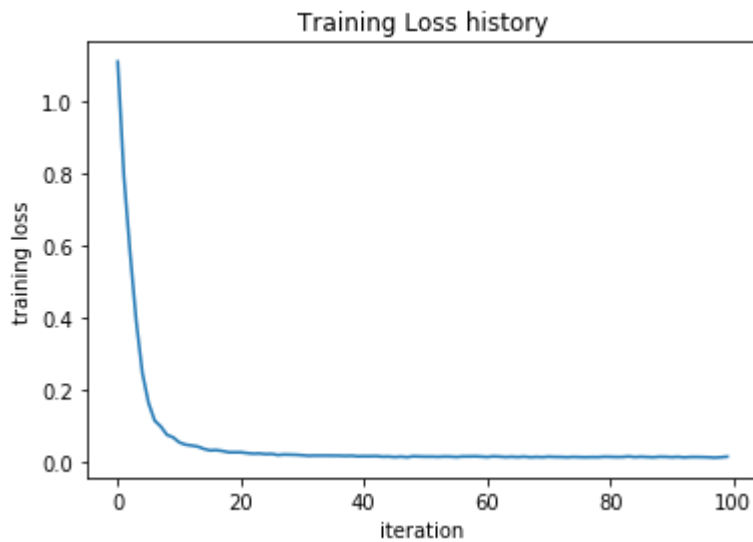
Implement `neural_net.train()` to train the network via stochastic gradient descent, much like the softmax and SVM.

```
In [161]: 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.

```
In [162]: 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%.

```
In [187]: 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 2.3027659478724885
iteration 100 / 1000: loss 2.301882808207999
iteration 200 / 1000: loss 2.298416283890018
iteration 300 / 1000: loss 2.2771642888374384
iteration 400 / 1000: loss 2.2239361709076495
iteration 500 / 1000: loss 2.1318780215963495
iteration 600 / 1000: loss 1.9862680393440117
iteration 700 / 1000: loss 2.067437711434443
iteration 800 / 1000: loss 1.9510541726833204
iteration 900 / 1000: loss 1.9645401525616222
Validation accuracy: 0.28
```

## 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 [168]: stats['train_acc_history']
```

```
Out[168]: [0.095000000000000001,
0.14499999999999999,
0.23000000000000001,
0.27000000000000002,
0.27500000000000002]
```

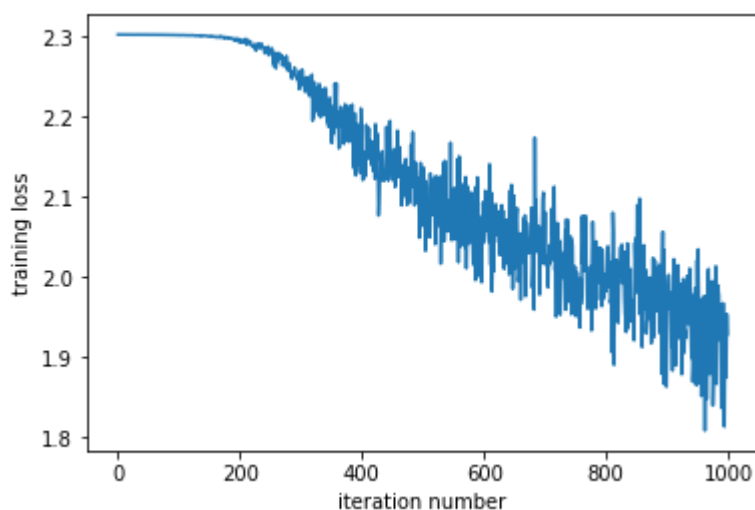
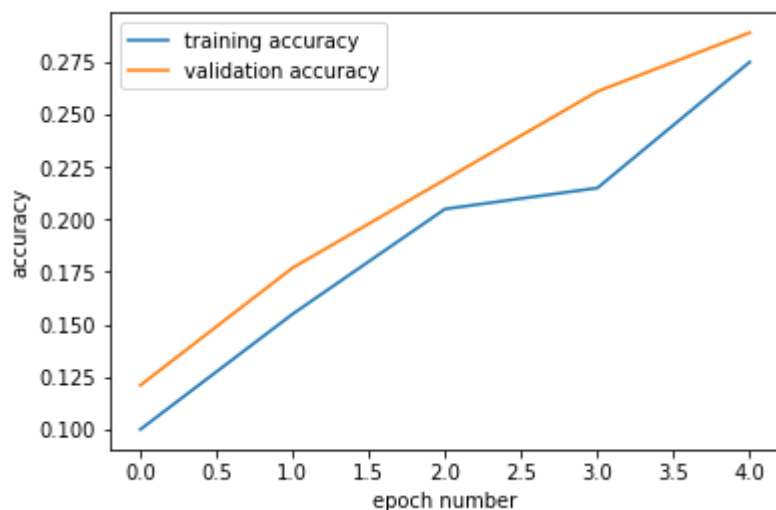
```
In [186]: # ===== #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ===== #

# Plot the loss function and train / validation accuracies

plt.figure(1)
plt.plot(stats['train_acc_history'])
plt.plot(stats['val_acc_history'])
plt.xlabel('epoch number')
plt.ylabel('accuracy')
plt.legend(['training accuracy', 'validation accuracy'])
plt.show

plt.figure(2)
plt.plot(stats['loss_history'])
plt.xlabel('iteration number')
plt.ylabel('training loss')
plt.show()

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



## Answers:

(1) From the above graphs, we can see a few reasons for why the training accuracy isn't great.

Firstly, we can see that there are not enough iterations to arrive to a local minimum. The training loss is still in a negative slope as a function of the iteration number. Thus, we can expect that by increasing the number of iterations, the training loss would decrease even more.

Secondly, we can see that the training rate is quite low. Thus, increasing the learning rate might help speeding up the training process.

(2) As stated before, we should increase the number of iterations, and possibly increase the learning rate.

## 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 [196]: 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)!
# ===== #
best_net = TwoLayerNet(input_size, hidden_size, num_classes)

stats = best_net.train(X_train, y_train, X_val, y_val,
                       num_iters=12000, batch_size=200,
                       learning_rate=4e-4, learning_rate_decay=0.95,
                       reg=0.3, verbose=True)

# ===== #
# END YOUR CODE HERE
# ===== #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 12000: loss 2.3027983384622157
iteration 100 / 12000: loss 2.2112562799492284
iteration 200 / 12000: loss 2.005117009546034
iteration 300 / 12000: loss 1.8807443921643985
iteration 400 / 12000: loss 1.8602474705845493
iteration 500 / 12000: loss 1.7533035079493708
iteration 600 / 12000: loss 1.7551234730652
iteration 700 / 12000: loss 1.6158704103571528
iteration 800 / 12000: loss 1.5807352193590452
iteration 900 / 12000: loss 1.6295480350826965
iteration 1000 / 12000: loss 1.635662107163549
iteration 1100 / 12000: loss 1.5402873017701144
iteration 1200 / 12000: loss 1.562718393466583
iteration 1300 / 12000: loss 1.5735948823340546
iteration 1400 / 12000: loss 1.579845416729201
iteration 1500 / 12000: loss 1.6995466344740662
iteration 1600 / 12000: loss 1.4917033088084017
iteration 1700 / 12000: loss 1.6466691518575056
iteration 1800 / 12000: loss 1.4639567098332584
iteration 1900 / 12000: loss 1.4917351915584778
iteration 2000 / 12000: loss 1.5362009996594839
iteration 2100 / 12000: loss 1.5320511286144076
iteration 2200 / 12000: loss 1.5289768178110181
iteration 2300 / 12000: loss 1.5942133377801646
iteration 2400 / 12000: loss 1.5655335546431963
iteration 2500 / 12000: loss 1.3706303397897748
iteration 2600 / 12000: loss 1.404434648344379
iteration 2700 / 12000: loss 1.429482924664336
iteration 2800 / 12000: loss 1.5766967093416606
iteration 2900 / 12000: loss 1.4286978927004843
iteration 3000 / 12000: loss 1.4458285579050347
iteration 3100 / 12000: loss 1.4699408378619392
iteration 3200 / 12000: loss 1.4283418742361669
iteration 3300 / 12000: loss 1.4165844532742609
iteration 3400 / 12000: loss 1.4048765800560055
iteration 3500 / 12000: loss 1.4274020490786252
iteration 3600 / 12000: loss 1.5238854235180175
iteration 3700 / 12000: loss 1.3918327901515357
iteration 3800 / 12000: loss 1.48308904943206
iteration 3900 / 12000: loss 1.4739234918520323
iteration 4000 / 12000: loss 1.3771865029733812
iteration 4100 / 12000: loss 1.3887235933304416
iteration 4200 / 12000: loss 1.4873751128365873
iteration 4300 / 12000: loss 1.3878438494325358
iteration 4400 / 12000: loss 1.4079495713032404
iteration 4500 / 12000: loss 1.3928161085808455
iteration 4600 / 12000: loss 1.3058586032779045
iteration 4700 / 12000: loss 1.3316754376800843
iteration 4800 / 12000: loss 1.355626096779089
iteration 4900 / 12000: loss 1.2818108057280178
iteration 5000 / 12000: loss 1.338209639216842
iteration 5100 / 12000: loss 1.5611458565606386
iteration 5200 / 12000: loss 1.4038471658995737
iteration 5300 / 12000: loss 1.4233515021963035
iteration 5400 / 12000: loss 1.4430413919272178
iteration 5500 / 12000: loss 1.3207571998147893
iteration 5600 / 12000: loss 1.326185919655488
```

```
iteration 5700 / 12000: loss 1.3879925255594008
iteration 5800 / 12000: loss 1.3947350605552071
iteration 5900 / 12000: loss 1.4965013334298354
iteration 6000 / 12000: loss 1.3735927269659962
iteration 6100 / 12000: loss 1.3449182943741498
iteration 6200 / 12000: loss 1.2821050543299548
iteration 6300 / 12000: loss 1.5001258286637384
iteration 6400 / 12000: loss 1.328587138299772
iteration 6500 / 12000: loss 1.3438559711425337
iteration 6600 / 12000: loss 1.4584467122870557
iteration 6700 / 12000: loss 1.218421900737831
iteration 6800 / 12000: loss 1.3840877662754987
iteration 6900 / 12000: loss 1.3559356275576746
iteration 7000 / 12000: loss 1.3427177733138635
iteration 7100 / 12000: loss 1.3308309427571676
iteration 7200 / 12000: loss 1.3428592919630804
iteration 7300 / 12000: loss 1.2661058878862992
iteration 7400 / 12000: loss 1.3407682856666059
iteration 7500 / 12000: loss 1.2613548402185295
iteration 7600 / 12000: loss 1.3868462701643032
iteration 7700 / 12000: loss 1.3991474600221463
iteration 7800 / 12000: loss 1.3499007138545303
iteration 7900 / 12000: loss 1.3414295480311613
iteration 8000 / 12000: loss 1.4528323187013048
iteration 8100 / 12000: loss 1.417464011363316
iteration 8200 / 12000: loss 1.3080401696341628
iteration 8300 / 12000: loss 1.2597398216376625
iteration 8400 / 12000: loss 1.3849987313056942
iteration 8500 / 12000: loss 1.2926882003210083
iteration 8600 / 12000: loss 1.2189182696643983
iteration 8700 / 12000: loss 1.3162236134303529
iteration 8800 / 12000: loss 1.394068400870541
iteration 8900 / 12000: loss 1.1934476584459062
iteration 9000 / 12000: loss 1.1928362388126335
iteration 9100 / 12000: loss 1.3255108313298318
iteration 9200 / 12000: loss 1.4328982200706837
iteration 9300 / 12000: loss 1.2548171712629015
iteration 9400 / 12000: loss 1.4831173063036194
iteration 9500 / 12000: loss 1.3365881403620548
iteration 9600 / 12000: loss 1.2871459196175377
iteration 9700 / 12000: loss 1.3615420485064684
iteration 9800 / 12000: loss 1.392893347748091
iteration 9900 / 12000: loss 1.3305222881073764
iteration 10000 / 12000: loss 1.364783754992933
iteration 10100 / 12000: loss 1.4039960451685025
iteration 10200 / 12000: loss 1.4044244786678022
iteration 10300 / 12000: loss 1.2654311904056612
iteration 10400 / 12000: loss 1.3141728013526421
iteration 10500 / 12000: loss 1.1791312286422881
iteration 10600 / 12000: loss 1.4044137324460177
iteration 10700 / 12000: loss 1.324184490193817
iteration 10800 / 12000: loss 1.3208744488557642
iteration 10900 / 12000: loss 1.4115116122659837
iteration 11000 / 12000: loss 1.3755676698475092
iteration 11100 / 12000: loss 1.3642601678936825
iteration 11200 / 12000: loss 1.3422716246498752
iteration 11300 / 12000: loss 1.213152252821828
```

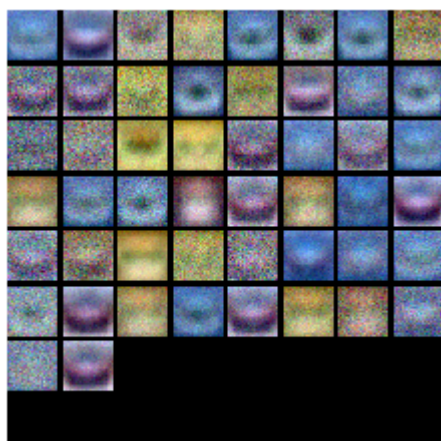
```
iteration 11400 / 12000: loss 1.3826560986668337
iteration 11500 / 12000: loss 1.3784164495406177
iteration 11600 / 12000: loss 1.414647220534285
iteration 11700 / 12000: loss 1.442284873382549
iteration 11800 / 12000: loss 1.319430273731646
iteration 11900 / 12000: loss 1.492905465817912
Validation accuracy: 0.504
```

```
In [197]: 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 suboptimal net's weights associated with each neuron are either very noisy or very similar to each other. This suggests that the hidden layer's neurons either learned nothing useful, or learned to produce similar features as the other neurons. This in turn doesn't make the model robust, because a successful model requires a "good" feature construction/learning.

On the other hand, we can see that the best net's weights associated with each neuron are very diverse, crisp, and informative (We can barely see noisy neurons). This suggests that the neurons of the hidden layer successfully constructed useful features from the inputs, which makes the model generic and useful.

## Evaluate on test set

```
In [198]: test_acc = (best_net.predict(X_test) == y_test).mean()  
          print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.541
```

## neural\_net.py

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  """
5  This code was originally written for CS 231n at Stanford University
6  (cs231n.stanford.edu). It has been modified in various areas for use in the
7  ECE 239AS class at UCLA. This includes the descriptions of what code to
8  implement as well as some slight potential changes in variable names to be
9  consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
10 permission to use this code. To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14 class TwoLayerNet(object):
15     """
16     A two-layer fully-connected neural network. The net has an input dimension of
17     N, a hidden layer dimension of H, and performs classification over C classes.
18     We train the network with a softmax loss function and L2 regularization on the
19     weight matrices. The network uses a ReLU nonlinearity after the first fully
20     connected layer.
21
22     In other words, the network has the following architecture:
23
24     input - fully connected layer - ReLU - fully connected layer - softmax
25
26     The outputs of the second fully-connected layer are the scores for each class.
27     """
28
29     def __init__(self, input_size, hidden_size, output_size, std=1e-4):
30         """
31         Initialize the model. Weights are initialized to small random values and
32         biases are initialized to zero. Weights and biases are stored in the
33         variable self.params, which is a dictionary with the following keys:
34
35         W1: First layer weights; has shape (H, D)
36         b1: First layer biases; has shape (H,)
37         W2: Second layer weights; has shape (C, H)
38         b2: Second layer biases; has shape (C,)
39
40         Inputs:
41         - input_size: The dimension D of the input data.
42         - hidden_size: The number of neurons H in the hidden layer.
43         - output_size: The number of classes C.
44         """
45         self.params = {}
46         self.params['W1'] = std * np.random.randn(hidden_size, input_size)
47         self.params['b1'] = np.zeros(hidden_size)
48         self.params['W2'] = std * np.random.randn(output_size, hidden_size)
49         self.params['b2'] = np.zeros(output_size)
50
51
52     def loss(self, X, y=None, reg=0.0):
53         """
54         Compute the loss and gradients for a two layer fully connected neural
55         network.
56
57         Inputs:
58         - X: Input data of shape (N, D). Each X[i] is a training sample.
59         - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
60           an integer in the range 0 <= y[i] < C. This parameter is optional; if it
61           is not passed then we only return scores, and if it is passed then we
62           instead return the loss and gradients.
63         - reg: Regularization strength.
64
65         Returns:
66         If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
67         the score for class c on input X[i].
68
69         If y is not None, instead return a tuple of:
70         - loss: Loss (data loss and regularization loss) for this batch of training
71           samples.
72         - grads: Dictionary mapping parameter names to gradients of those parameters
73           with respect to the loss function; has the same keys as self.params.
74         """
75         # Unpack variables from the params dictionary
76         W1, b1 = self.params['W1'], self.params['b1']
77         W2, b2 = self.params['W2'], self.params['b2']
78         N, D = X.shape
79
80         # Compute the forward pass
81         scores = None
82

```

```

83 # ===== #
84 # YOUR CODE HERE:
85 # Calculate the output scores of the neural network. The result
86 # should be (N, C). As stated in the description for this class,
87 # there should not be a ReLU layer after the second FC layer.
88 # The output of the second FC layer is the output scores. Do not
89 # use a for loop in your implementation.
90 # ===== #
91
92 def softmax(x):
93     e_x = np.exp(x - np.max(x))
94     return e_x / e_x.sum()
95
96 XdotW1_T = X.dot(W1.T)
97 perceptron_1_out = XdotW1_T + b1
98 layer_1_out = perceptron_1_out * (perceptron_1_out > 0)
99 scores = layer_1_out.dot(W2.T) + b2
100
101 # ===== #
102 # END YOUR CODE HERE
103 # ===== #
104
105
106 # If the targets are not given then jump out, we're done
107 if y is None:
108     return scores
109
110 # Compute the loss
111 loss = None
112
113 # ===== #
114 # YOUR CODE HERE:
115 # Calculate the loss of the neural network. This includes the
116 # softmax loss and the L2 regularization for W1 and W2. Store the
117 # total loss in the variable loss. Multiply the regularization
118 # loss by 0.5 (in addition to the factor reg).
119 # ===== #
120
121 # scores is num_examples by num_classes
122 W1_norm_sqrd = (1/2) * (np.linalg.norm(W1) ** 2)
123 W2_norm_sqrd = (1/2) * (np.linalg.norm(W2) ** 2)
124 minibatch_size = scores.shape[0]
125 loss = (1 / float(minibatch_size)) * np.sum(
126     np.log(np.sum(np.exp(scores.T), axis=0)) - np.choose(y, scores.T)) + reg * (W1_norm_sqrd + W2_norm_sqrd)
127 # ===== #
128 # END YOUR CODE HERE
129 # ===== #
130
131 grads = {}
132
133 # ===== #
134 # YOUR CODE HERE:
135 # Implement the backward pass. Compute the derivatives of the
136 # weights and the biases. Store the results in the grads
137 # dictionary. e.g., grads['W1'] should store the gradient for
138 # W1, and be of the same size as W1.
139 # ===== #
140
141 ...
142 Calculation of softmax gradient with respect to the scores of the last layer. The code is taken from the previous
143 homework, with slight modifications
144 ...
145
146 softmax_nominators = np.exp(scores.T - np.amax(scores.T, axis=0))
147 softmax_matrix = softmax_nominators / np.sum(softmax_nominators, axis=0)
148 softmax_matrix[y, np.arange(N)] -= 1
149 softmax_grad = (1 / N) * softmax_matrix
150
151 grads['b2'] = np.sum(softmax_grad, axis=1)
152 grads['W2'] = softmax_grad.dot(layer_1_out) + reg * W2
153 relu_activations = np.array([1] * (perceptron_1_out > 0))
154 grad_bef_relu = W2.T.dot(softmax_grad) * relu_activations.T
155 grads['b1'] = np.sum(grad_bef_relu, axis=1)
156 grads['W1'] = grad_bef_relu.dot(X) + reg * W1
157
158
159 # ===== #
160 # END YOUR CODE HERE
161 # ===== #
162
163 return loss, grads
164
165 def train(self, X, y, X_val, y_val,
166         learning_rate=1e-3, learning_rate_decay=0.95,
167         reg=1e-5, num_iters=100,

```

```

168         batch_size=200, verbose=False):
169     """
170     Train this neural network using stochastic gradient descent.
171
172     Inputs:
173     - X: A numpy array of shape (N, D) giving training data.
174     - y: A numpy array of shape (N,) giving training labels; y[i] = c means that
175       X[i] has label c, where 0 ≤ c < C.
176     - X_val: A numpy array of shape (N_val, D) giving validation data.
177     - y_val: A numpy array of shape (N_val,) giving validation labels.
178     - learning_rate: Scalar giving learning rate for optimization.
179     - learning_rate_decay: Scalar giving factor used to decay the learning rate
180       after each epoch.
181     - reg: Scalar giving regularization strength.
182     - num_iters: Number of steps to take when optimizing.
183     - batch_size: Number of training examples to use per step.
184     - verbose: boolean; if true print progress during optimization.
185     """
186     num_train = X.shape[0]
187     iterations_per_epoch = max(num_train / batch_size, 1)
188
189     # Use SGD to optimize the parameters in self.model
190     loss_history = []
191     train_acc_history = []
192     val_acc_history = []
193
194     for it in np.arange(num_iters):
195         X_batch = None
196         y_batch = None
197
198         # ===== #
199         # YOUR CODE HERE:
200         # Create a minibatch by sampling batch_size samples randomly.
201         # ===== #
202         idx = np.random.randint(low=0, high=X.shape[0], size=batch_size)
203         X_batch = X[idx]
204         y_batch = y[idx]
205
206         # ===== #
207         # END YOUR CODE HERE
208         # ===== #
209
210         # Compute loss and gradients using the current minibatch
211         loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
212         loss_history.append(loss)
213
214         # ===== #
215         # YOUR CODE HERE:
216         # Perform a gradient descent step using the minibatch to update
217         # all parameters (i.e., W1, W2, b1, and b2).
218         # ===== #
219
220         self.params['W1'] -= learning_rate * grads['W1']
221         self.params['b1'] -= learning_rate * grads['b1']
222         self.params['W2'] -= learning_rate * grads['W2']
223         self.params['b2'] -= learning_rate * grads['b2']
224
225         # ===== #
226         # END YOUR CODE HERE
227         # ===== #
228
229         if verbose and it % 100 == 0:
230             print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
231
232         # Every epoch, check train and val accuracy and decay learning rate.
233         if it % iterations_per_epoch == 0:
234             # Check accuracy
235             train_acc = (self.predict(X_batch) == y_batch).mean()
236             val_acc = (self.predict(X_val) == y_val).mean()
237             train_acc_history.append(train_acc)
238             val_acc_history.append(val_acc)
239
240             # Decay learning rate
241             learning_rate *= learning_rate_decay
242
243     return {
244         'loss_history': loss_history,
245         'train_acc_history': train_acc_history,
246         'val_acc_history': val_acc_history,
247     }
248
249     def predict(self, X):
250         """
251         Use the trained weights of this two-layer network to predict labels for
252         data points. For each data point we predict scores for each of the C

```

```
253     classes, and assign each data point to the class with the highest score.
254
255     Inputs:
256     - X: A numpy array of shape (N, D) giving N D-dimensional data points to
257       classify.
258
259     Returns:
260     - y_pred: A numpy array of shape (N,) giving predicted labels for each of
261       the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
262       to have class c, where  $0 \leq c < C$ .
263     """
264     y_pred = None
265
266     # ===== #
267     # YOUR CODE HERE:
268     #   Predict the class given the input data.
269     # ===== #
270     scores = self.loss(X)
271     y_pred = np.argmax(scores, axis=1)
272
273
274     # ===== #
275     # END YOUR CODE HERE
276     # ===== #
277
278     return y_pred
279
280
281
```

## **2 General FC neural network**

## 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](http://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
```

```
In [1]: ## 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))))
```

```
In [5]: # Load the (preprocessed) CIFAR10 data.

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

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

## 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.

```
In [6]: # 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: 9.769848888397517e-10
```

## Affine layer backward pass

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

```
In [16]: # 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: 8.91750613938316e-10
dw error: 2.939107482755546e-10
db error: 1.9058986817585e-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.

```
In [17]: # 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,  0.136
36364,],
                        [ 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.

```
In [19]: 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: 3.2755857297141065e-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.

```
In [20]: 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: 2.108073174399626e-10
dw error: 2.1813229897588312e-08
db error: 7.826601920934216e-12
```

## 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.

```

In [21]: 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: 8.998661191083356
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.302451635416486
dx error: 7.69323964766518e-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.

```

In [41]: 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, 14.5719843
4, 15.33206765, 16.09215096],
    [12.05769098, 12.74614105, 13.43459113, 14.1230412, 14.8114912
8, 15.49994135, 16.18839143],
    [12.58373087, 13.20054771, 13.81736455, 14.43418138, 15.0509982
2, 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 l
oss'

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: 1.833656129733658e-08

W2 relative error: 3.3727467137085415e-10

b1 relative error: 8.008665519199494e-09

b2 relative error: 2.5307826250578787e-10

Running numeric gradient check with reg = 0.7

W1 relative error: 2.527915286171985e-07

W2 relative error: 2.8508510893102143e-08

b1 relative error: 1.3467618820476118e-08

b2 relative error: 1.968057921260679e-09

## 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%.

```
In [48]: 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 50%. We won't have you optimize this further
#   since you did it in the previous notebook.
#
# ===== #

model = TwoLayerNet(hidden_dims=200, reg=0.3)
solver = Solver(model, data, update_rule='sgd', optim_config={'learning_rate':4e-4},
                 lr_decay=0.95, batch_size=200, num_epochs=10, print_every=100)
solver.train()

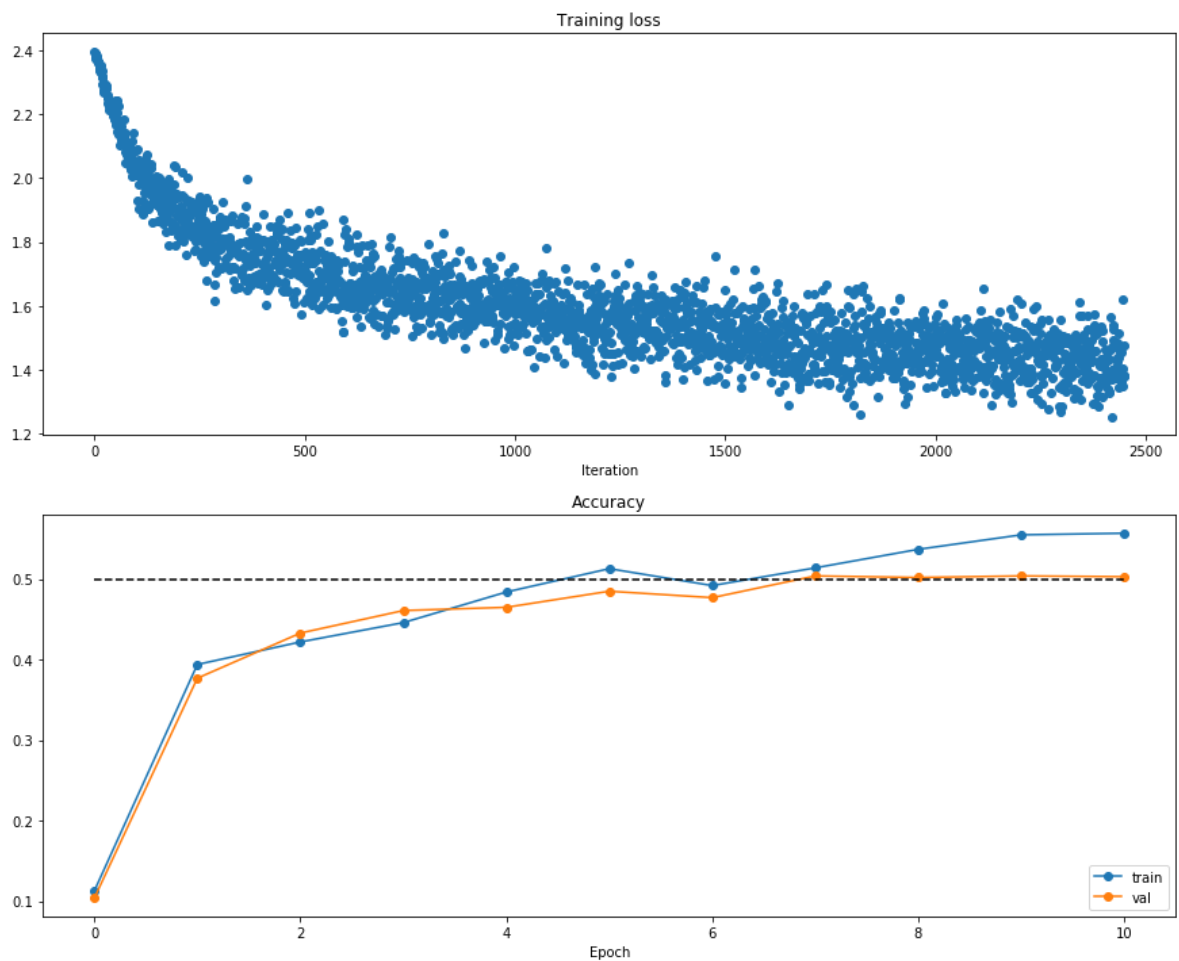
# ===== #
# END YOUR CODE HERE
# ===== #
```

```
(Iteration 1 / 2450) loss: 2.396027
(Epoch 0 / 10) train acc: 0.113000; val_acc: 0.104000
(Iteration 101 / 2450) loss: 2.009685
(Iteration 201 / 2450) loss: 1.839928
(Epoch 1 / 10) train acc: 0.394000; val_acc: 0.377000
(Iteration 301 / 2450) loss: 1.775627
(Iteration 401 / 2450) loss: 1.889422
(Epoch 2 / 10) train acc: 0.422000; val_acc: 0.433000
(Iteration 501 / 2450) loss: 1.707768
(Iteration 601 / 2450) loss: 1.656731
(Iteration 701 / 2450) loss: 1.786520
(Epoch 3 / 10) train acc: 0.446000; val_acc: 0.461000
(Iteration 801 / 2450) loss: 1.508494
(Iteration 901 / 2450) loss: 1.662802
(Epoch 4 / 10) train acc: 0.484000; val_acc: 0.465000
(Iteration 1001 / 2450) loss: 1.610448
(Iteration 1101 / 2450) loss: 1.562393
(Iteration 1201 / 2450) loss: 1.670454
(Epoch 5 / 10) train acc: 0.513000; val_acc: 0.485000
(Iteration 1301 / 2450) loss: 1.443475
(Iteration 1401 / 2450) loss: 1.371782
(Epoch 6 / 10) train acc: 0.492000; val_acc: 0.477000
(Iteration 1501 / 2450) loss: 1.648775
(Iteration 1601 / 2450) loss: 1.560411
(Iteration 1701 / 2450) loss: 1.465578
(Epoch 7 / 10) train acc: 0.514000; val_acc: 0.504000
(Iteration 1801 / 2450) loss: 1.493812
(Iteration 1901 / 2450) loss: 1.585204
(Epoch 8 / 10) train acc: 0.537000; val_acc: 0.502000
(Iteration 2001 / 2450) loss: 1.462025
(Iteration 2101 / 2450) loss: 1.412057
(Iteration 2201 / 2450) loss: 1.433635
(Epoch 9 / 10) train acc: 0.555000; val_acc: 0.504000
(Iteration 2301 / 2450) loss: 1.491427
(Iteration 2401 / 2450) loss: 1.425856
(Epoch 10 / 10) train acc: 0.557000; val_acc: 0.503000
```

In [49]: *# 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.

```
In [60]: 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.float
64)

    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, gra
ds[name])))

Running check with reg = 0
Initial loss: 2.311057943012428
W1 relative error: 1.7465988060536975e-06
W2 relative error: 7.661663873944155e-08
W3 relative error: 2.244179132925438e-07
b1 relative error: 5.516366127912843e-08
b2 relative error: 4.111517127567695e-09
b3 relative error: 1.4560303049505286e-10
Running check with reg = 3.14
Initial loss: 6.99748972265839
W1 relative error: 1.5077058713280157e-08
W2 relative error: 1.6703938600294386e-08
W3 relative error: 2.1003550557039997e-08
b1 relative error: 5.473224952160955e-09
b2 relative error: 8.500753795678266e-09
b3 relative error: 2.003082213907742e-10
```

```
In [65]: # Use the three layer neural network to overfit a small dataset.

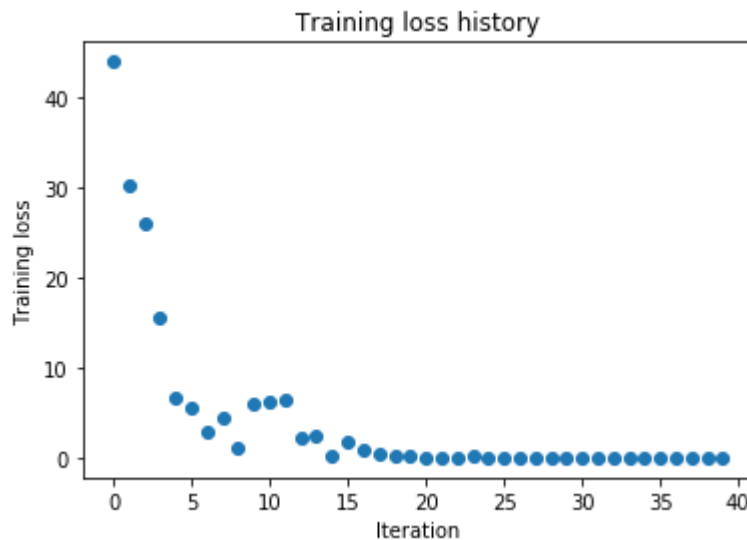
num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

#### !!!!!!!
# 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 = 6e-2
learning_rate = 4e-4

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': learning_rate,
                 })
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: 44.066522
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.132000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.125000
(Epoch 2 / 20) train acc: 0.420000; val_acc: 0.118000
(Epoch 3 / 20) train acc: 0.560000; val_acc: 0.122000
(Epoch 4 / 20) train acc: 0.700000; val_acc: 0.132000
(Epoch 5 / 20) train acc: 0.700000; val_acc: 0.134000
(Iteration 11 / 40) loss: 6.186395
(Epoch 6 / 20) train acc: 0.760000; val_acc: 0.127000
(Epoch 7 / 20) train acc: 0.840000; val_acc: 0.128000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.143000
(Epoch 9 / 20) train acc: 0.880000; val_acc: 0.147000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.143000
(Iteration 21 / 40) loss: 0.007950
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.143000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.145000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.145000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.145000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.145000
(Iteration 31 / 40) loss: 0.002341
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.145000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.143000
```



## fc\_net.py

```

1  import numpy as np
2
3  from .layers import *
4  from .layer_utils import *
5
6  """
7  This code was originally written for CS 231n at Stanford University
8  (cs231n.stanford.edu). It has been modified in various areas for use in the
9  ECE 239AS class at UCLA. This includes the descriptions of what code to
10 implement as well as some slight potential changes in variable names to be
11 consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
12 permission to use this code. To see the original version, please visit
13 cs231n.stanford.edu.
14 """
15
16 class TwoLayerNet(object):
17     """
18     A two-layer fully-connected neural network with ReLU nonlinearity and
19     softmax loss that uses a modular layer design. We assume an input dimension
20     of D, a hidden dimension of H, and perform classification over C classes.
21
22     The architecture should be affine - relu - affine - softmax.
23
24     Note that this class does not implement gradient descent; instead, it
25     will interact with a separate Solver object that is responsible for running
26     optimization.
27
28     The learnable parameters of the model are stored in the dictionary
29     self.params that maps parameter names to numpy arrays.
30     """
31
32     def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
33                 dropout=0, weight_scale=1e-3, reg=0.0):
34         """
35         Initialize a new network.
36
37         Inputs:
38         - input_dim: An integer giving the size of the input
39         - hidden_dims: An integer giving the size of the hidden layer
40         - num_classes: An integer giving the number of classes to classify
41         - dropout: Scalar between 0 and 1 giving dropout strength.
42         - weight_scale: Scalar giving the standard deviation for random
43           initialization of the weights.
44         - reg: Scalar giving L2 regularization strength.
45         """
46         self.params = {}
47         self.reg = reg
48
49         # ===== #
50         # YOUR CODE HERE:
51         # Initialize W1, W2, b1, and b2. Store these as self.params['W1'],
52         # self.params['W2'], self.params['b1'] and self.params['b2']. The
53         # biases are initialized to zero and the weights are initialized
54         # so that each parameter has mean 0 and standard deviation weight_scale.
55         # The dimensions of W1 should be (input_dim, hidden_dim) and the
56         # dimensions of W2 should be (hidden_dims, num_classes)
57         # ===== #
58
59         self.params['W1'] = weight_scale * np.random.randn(input_dim, hidden_dims)
60         self.params['b1'] = np.zeros(hidden_dims)
61         self.params['W2'] = weight_scale * np.random.randn(hidden_dims, num_classes)
62         self.params['b2'] = np.zeros(num_classes)
63
64         # ===== #
65         # END YOUR CODE HERE
66         # ===== #
67
68     def loss(self, X, y=None):
69         """
70         Compute loss and gradient for a minibatch of data.
71
72         Inputs:
73         - X: Array of input data of shape (N, d_1, ..., d_k)
74         - y: Array of labels, of shape (N,). y[i] gives the label for X[i].
75
76         Returns:
77         If y is None, then run a test-time forward pass of the model and return:
78         - scores: Array of shape (N, C) giving classification scores, where
79           scores[i, c] is the classification score for X[i] and class c.

```

```

80
81     If y is not None, then run a training-time forward and backward pass and
82     return a tuple of:
83     - loss: Scalar value giving the loss
84     - grads: Dictionary with the same keys as self.params, mapping parameter
85     names to gradients of the loss with respect to those parameters.
86     """
87     scores = None
88
89     # ===== #
90     # YOUR CODE HERE:
91     # Implement the forward pass of the two-layer neural network. Store
92     # the class scores as the variable 'scores'. Be sure to use the layers
93     # you prior implemented.
94     # ===== #
95
96     layer_1_out, layer_1_cache = affine_relu_forward(np.array(X), self.params['W1'], self.params['b1'])
97     scores, layer_2_cache = affine_forward(np.array(layer_1_out), self.params['W2'], self.params['b2'])
98
99     # ===== #
100    # END YOUR CODE HERE
101    # ===== #
102
103    # If y is None then we are in test mode so just return scores
104    if y is None:
105        return scores
106
107    loss, grads = 0, {}
108    # ===== #
109    # YOUR CODE HERE:
110    # Implement the backward pass of the two-layer neural net. Store
111    # the loss as the variable 'loss' and store the gradients in the
112    # 'grads' dictionary. For the grads dictionary, grads['W1'] holds
113    # the gradient for W1, grads['b1'] holds the gradient for b1, etc.
114    # i.e., grads[k] holds the gradient for self.params[k].
115    #
116    # Add L2 regularization, where there is an added cost  $0.5 * \text{self.reg} * W^2$ 
117    # for each W. Be sure to include the 0.5 multiplying factor to
118    # match our implementation.
119    #
120    # And be sure to use the layers you prior implemented.
121    # ===== #
122
123    W1_norm_sqrd = (np.linalg.norm(self.params['W1']) ** 2)
124    W2_norm_sqrd = (np.linalg.norm(self.params['W2']) ** 2)
125
126    loss, soft_grad = softmax_loss(scores, y)
127    loss += 0.5 * self.reg * (W1_norm_sqrd + W2_norm_sqrd)
128    layer_2_back_grad, grads['W2'], grads['b2'] = affine_backward(soft_grad, layer_2_cache)
129    layer_1_back_grad, grads['W1'], grads['b1'] = affine_relu_backward(layer_2_back_grad, layer_1_cache)
130
131    grads['W2'] += self.reg * self.params['W2']
132    grads['W1'] += self.reg * self.params['W1']
133    # ===== #
134    # END YOUR CODE HERE
135    # ===== #
136
137    return loss, grads
138
139
140    class FullyConnectedNet(object):
141        """
142        A fully-connected neural network with an arbitrary number of hidden layers,
143        ReLU nonlinearities, and a softmax loss function. This will also implement
144        dropout and batch normalization as options. For a network with L layers,
145        the architecture will be
146
147        {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax
148
149        where batch normalization and dropout are optional, and the {...} block is
150        repeated L - 1 times.
151
152        Similar to the TwoLayerNet above, learnable parameters are stored in the
153        self.params dictionary and will be learned using the Solver class.
154        """
155
156        def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
157                    dropout=0, use_batchnorm=False, reg=0.0,
158                    weight_scale=1e-2, dtype=np.float32, seed=None):
159            """
160            Initialize a new FullyConnectedNet.
161

```

```

162 Inputs:
163 - hidden_dims: A list of integers giving the size of each hidden layer.
164 - input_dim: An integer giving the size of the input.
165 - num_classes: An integer giving the number of classes to classify.
166 - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
167   the network should not use dropout at all.
168 - use_batchnorm: Whether or not the network should use batch normalization.
169 - reg: Scalar giving L2 regularization strength.
170 - weight_scale: Scalar giving the standard deviation for random
171   initialization of the weights.
172 - dtype: A numpy datatype object; all computations will be performed using
173   this datatype. float32 is faster but less accurate, so you should use
174   float64 for numeric gradient checking.
175 - seed: If not None, then pass this random seed to the dropout layers. This
176   will make the dropout layers deterministic so we can gradient check the
177   model.
178 """
179 self.use_batchnorm = use_batchnorm
180 self.use_dropout = dropout > 0
181 self.reg = reg
182 self.num_layers = 1 + len(hidden_dims)
183 self.dtype = dtype
184 self.params = {}
185
186 # ===== #
187 # YOUR CODE HERE:
188 # Initialize all parameters of the network in the self.params dictionary.
189 # The weights and biases of layer 1 are W1 and b1; and in general the
190 # weights and biases of layer i are Wi and bi. The
191 # biases are initialized to zero and the weights are initialized
192 # so that each parameter has mean 0 and standard deviation weight_scale.
193 # ===== #
194
195 next_layer_input_dim = input_dim
196 for i, hidden_dim in enumerate(hidden_dims, start=1):
197     self.params['W'+str(i)] = weight_scale * np.random.randn(next_layer_input_dim, hidden_dim)
198     self.params['b'+str(i)] = np.zeros(hidden_dim)
199     next_layer_input_dim = hidden_dim
200
201 self.params['W'+str(self.num_layers)] = weight_scale * np.random.randn(next_layer_input_dim, num_classes)
202 self.params['b'+str(self.num_layers)] = np.zeros(num_classes)
203
204 # ===== #
205 # END YOUR CODE HERE
206 # ===== #
207
208 # When using dropout we need to pass a dropout_param dictionary to each
209 # dropout layer so that the layer knows the dropout probability and the mode
210 # (train / test). You can pass the same dropout_param to each dropout layer.
211 self.dropout_param = {}
212 if self.use_dropout:
213     self.dropout_param = {'mode': 'train', 'p': dropout}
214     if seed is not None:
215         self.dropout_param['seed'] = seed
216
217 # With batch normalization we need to keep track of running means and
218 # variances, so we need to pass a special bn_param object to each batch
219 # normalization layer. You should pass self.bn_params[0] to the forward pass
220 # of the first batch normalization layer, self.bn_params[1] to the forward
221 # pass of the second batch normalization layer, etc.
222 self.bn_params = []
223 if self.use_batchnorm:
224     self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
225
226 # Cast all parameters to the correct datatype
227 for k, v in self.params.items():
228     self.params[k] = v.astype(dtype)
229
230
231 def loss(self, X, y=None):
232     """
233     Compute loss and gradient for the fully-connected net.
234
235     Input / output: Same as TwoLayerNet above.
236     """
237     X = X.astype(self.dtype)
238     mode = 'test' if y is None else 'train'
239
240     # Set train/test mode for batchnorm params and dropout param since they
241     # behave differently during training and testing.
242     if self.dropout_param is not None:
243         self.dropout_param['mode'] = mode

```

```

244     if self.use_batchnorm:
245         for bn_param in self.bn_params:
246             bn_param[mode] = mode
247
248     scores = None
249
250     # ===== #
251     # YOUR CODE HERE:
252     # Implement the forward pass of the FC net and store the output
253     # scores as the variable "scores".
254     # ===== #
255
256     caches = {}
257     layer_out = X
258     for i in range(1, self.num_layers):
259         layer_out, caches[i] = affine_relu_forward(np.array(layer_out),
260                                                     self.params['W'+str(i)],
261                                                     self.params['b'+str(i)])
262
263     scores, caches[self.num_layers] = affine_forward(np.array(layer_out),
264                                                       self.params['W'+str(self.num_layers)],
265                                                       self.params['b'+str(self.num_layers)])
266
267     # ===== #
268     # END YOUR CODE HERE
269     # ===== #
270
271     # If test mode return early
272     if mode == 'test':
273         return scores
274
275     loss, grads = 0.0, {}
276     # ===== #
277     # YOUR CODE HERE:
278     # Implement the backwards pass of the FC net and store the gradients
279     # in the grads dict, so that grads[k] is the gradient of self.params[k]
280     # Be sure your L2 regularization includes a 0.5 factor.
281     # ===== #
282     sum_of_W_norms = 0
283     for i in range(self.num_layers):
284         sum_of_W_norms += (np.linalg.norm(self.params['W'+str(i+1)]) ** 2)
285
286     loss, soft_grad = softmax_loss(scores, y)
287     loss += 0.5 * self.reg * sum_of_W_norms
288
289     layer_back_grad, grads['W'+str(self.num_layers)], grads['b'+str(self.num_layers)] = \
290         affine_backward(soft_grad, caches[self.num_layers])
291     grads['W' + str(self.num_layers)] += self.reg * self.params['W' + str(self.num_layers)]
292
293     for i in range(self.num_layers - 1, 0, -1):
294         layer_back_grad, grads['W'+str(i)], grads['b'+str(i)] = affine_relu_backward(layer_back_grad, caches[i])
295         grads['W' + str(i)] += self.reg * self.params['W' + str(i)]
296
297     # ===== #
298     # END YOUR CODE HERE
299     # ===== #
300     return loss, grads
301

```

## layers.py

```

1  import numpy as np
2  import pdb
3
4  """
5  This code was originally written for CS 231n at Stanford University
6  (cs231n.stanford.edu). It has been modified in various areas for use in the
7  ECE 239AS class at UCLA. This includes the descriptions of what code to
8  implement as well as some slight potential changes in variable names to be
9  consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
10 permission to use this code. To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14
15 def affine_forward(x, w, b):
16     """
17     Computes the forward pass for an affine (fully-connected) layer.
18
19     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
20     examples, where each example x[i] has shape (d_1, ..., d_k). We will
21     reshape each input into a vector of dimension  $\bar{D} = d_1 * \dots * d_k$ , and
22     then transform it to an output vector of dimension  $\bar{M}$ .
23
24     Inputs:
25     - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
26     - w: A numpy array of weights, of shape (D, M)
27     - b: A numpy array of biases, of shape (M,)
28
29     Returns a tuple of:
30     - out: output, of shape (N, M)
31     - cache: (x, w, b)
32     """
33
34     # ===== #
35     # YOUR CODE HERE:
36     # Calculate the output of the forward pass. Notice the dimensions
37     # of w are D x M, which is the transpose of what we did in earlier
38     # assignments.
39     # ===== #
40
41     out = x.reshape(x.shape[0], -1).dot(w) + b
42
43     # ===== #
44     # END YOUR CODE HERE
45     # ===== #
46
47     cache = (x, w, b)
48     return out, cache
49
50
51 def affine_backward(dout, cache):
52     """
53     Computes the backward pass for an affine layer.
54
55     Inputs:
56     - dout: Upstream derivative, of shape (N, M)
57     - cache: Tuple of:
58       - x: Input data, of shape (N, d_1, ... d_k)
59       - w: Weights, of shape (D, M)
60
61     Returns a tuple of:
62     - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
63     - dw: Gradient with respect to w, of shape (D, M)
64     - db: Gradient with respect to b, of shape (M,)
65     """
66     x, w, b = cache
67     dx, dw, db = None, None, None
68
69     # ===== #
70     # YOUR CODE HERE:
71     # Calculate the gradients for the backward pass.
72     # ===== #

```

```

73
74 # dout is N x M
75 # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M
76 # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshaping
77 # db should be M; it is just the sum over dout examples
78
79 db = np.sum(dout, axis=0)
80 dx = dout.dot(w.T).reshape(x.shape)
81 dw = x.reshape(x.shape[0], -1).T.dot(dout)
82 # ===== #
83 # END YOUR CODE HERE
84 # ===== #
85
86 return dx, dw, db
87
88 def relu_forward(x):
89     """
90     Computes the forward pass for a layer of rectified linear units (ReLU).
91
92     Input:
93     - x: Inputs, of any shape
94
95     Returns a tuple of:
96     - out: Output, of the same shape as x
97     - cache: x
98     """
99     # ===== #
100    # YOUR CODE HERE:
101    # Implement the ReLU forward pass.
102    # ===== #
103
104    out = x * (x > 0)
105    # ===== #
106    # END YOUR CODE HERE
107    # ===== #
108
109    cache = x
110    return out, cache
111
112
113 def relu_backward(dout, cache):
114     """
115     Computes the backward pass for a layer of rectified linear units (ReLU).
116
117     Input:
118     - dout: Upstream derivatives, of any shape
119     - cache: Input x, of same shape as dout
120
121     Returns:
122     - dx: Gradient with respect to x
123     """
124     x = cache
125
126     # ===== #
127     # YOUR CODE HERE:
128     # Implement the ReLU backward pass
129     # ===== #
130
131     # ReLU directs linearly to those > 0
132     dx = dout * (x > 0)
133
134     # ===== #
135     # END YOUR CODE HERE
136     # ===== #
137
138     return dx
139
140 def svm_loss(x, y):
141     """
142     Computes the loss and gradient using for multiclass SVM classification.
143
144     Inputs:
145     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
146         for the ith input.
147     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
148         0 <= y[i] < C

```

```

149
150 Returns a tuple of:
151 - loss: Scalar giving the loss
152 - dx: Gradient of the loss with respect to x
153 """
154 N = x.shape[0]
155 correct_class_scores = x[np.arange(N), y]
156 margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
157 margins[np.arange(N), y] = 0
158 loss = np.sum(margins) / N
159 num_pos = np.sum(margins > 0, axis=1)
160 dx = np.zeros_like(x)
161 dx[margins > 0] = 1
162 dx[np.arange(N), y] -= num_pos
163 dx /= N
164 return loss, dx
165
166
167 def softmax_loss(x, y):
168 """
169 Computes the loss and gradient for softmax classification.
170
171 Inputs:
172 - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
173    for the ith input.
174 - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
175    0 <= y[i] < C
176
177 Returns a tuple of:
178 - loss: Scalar giving the loss
179 - dx: Gradient of the loss with respect to x
180 """
181
182 probs = np.exp(x - np.max(x, axis=1, keepdims=True))
183 probs /= np.sum(probs, axis=1, keepdims=True)
184 N = x.shape[0]
185 loss = -np.sum(np.log(probs[np.arange(N), y])) / N
186 dx = probs.copy()
187 dx[np.arange(N), y] -= 1
188 dx /= N
189 return loss, dx
190

```