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.

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 ba
ckward pass.
# If your implementation is correct, the difference between the numeri
c 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(par
am_grad_num, grads[param_name])))
```

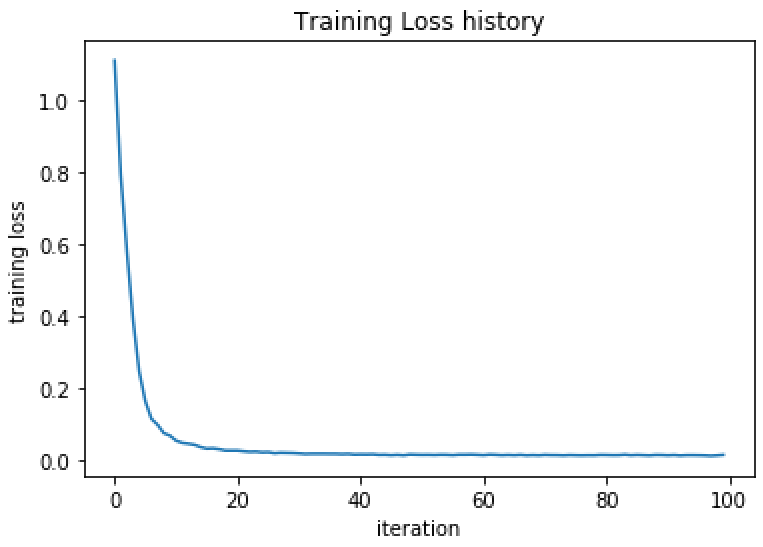
```
b1 max relative error: 3.172680092703762e-09
W2 max relative error: 3.425470383805365e-10
W1 max relative error: 1.2832784652838671e-09
b2 max relative error: 1.8391748601536041e-10
```

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 p
repare
    it for the two-layer neural net classifier. These are the same ste
ps 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.

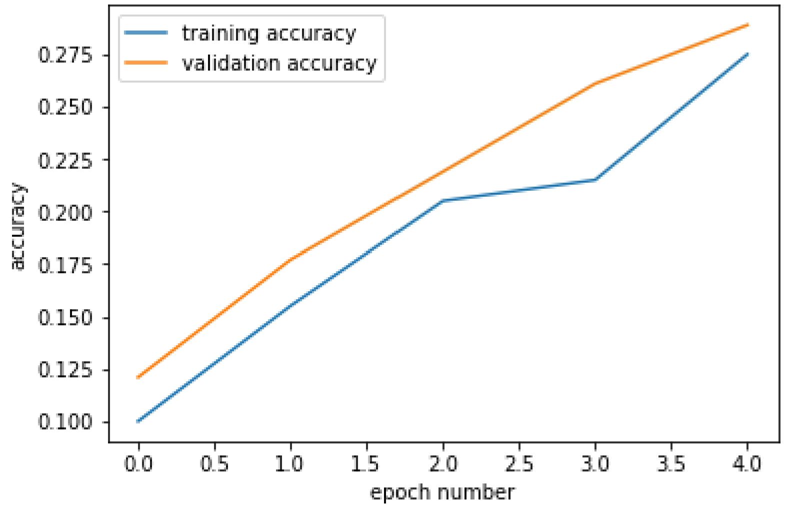
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

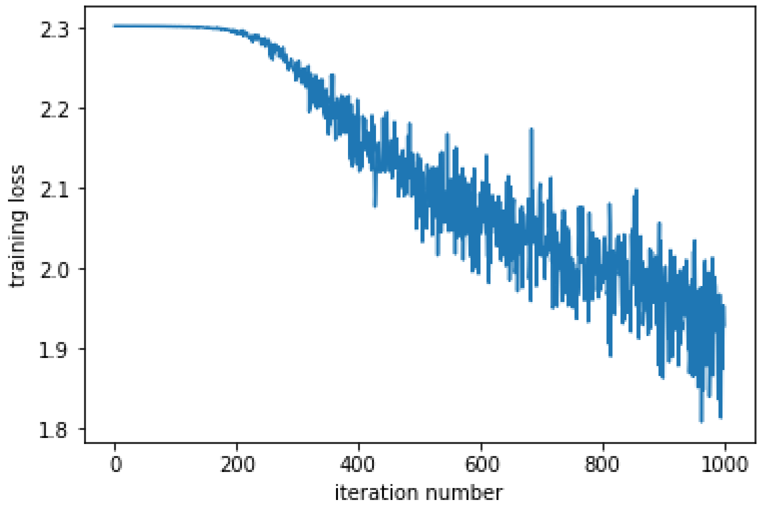
(2) How should you fix the problems you identified in (1)?

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

Out[168]:
```
[0.09500000000000001,
 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.

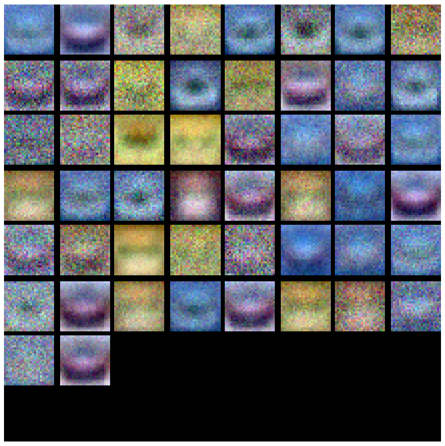
Optimize the neural network

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]:
```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 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
```