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 [2]:
```
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

In [3]:
```
from nndl.neural_net import TwoLayerNet
```

In [4]:
```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

In [5]:
```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.
#My debug code
print('Xshape: ', X.shape)
#End my debug code

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)))
```

```
Xshape:  (5, 4)
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:
3.381231233889892e-08
```

Forward pass loss

In [6]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

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

Backward pass

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

In [7]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward p
ass.
# 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)

#print(loss, grads)

# 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], verbos
e=False)
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_
num, grads[param_name])))
```

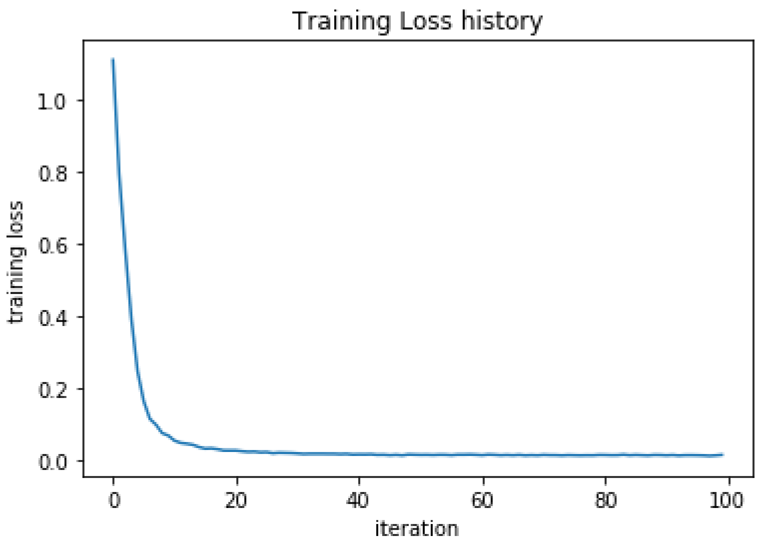
```
W2 max relative error: 2.9632227682005116e-10
b2 max relative error: 1.2482714253983918e-09
W1 max relative error: 1.2832823337649917e-09
b1 max relative error: 3.172680092703762e-09
```

Training the network

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

In [8]:
```
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:  0.014498406590265567
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [9]:
```
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 [10]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

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

```
iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302122329647926
iteration 200 / 1000: loss 2.2956767854707882
iteration 300 / 1000: loss 2.2523144504019696
iteration 400 / 1000: loss 2.1896338140489533
iteration 500 / 1000: loss 2.117053945819248
iteration 600 / 1000: loss 2.0653486572337925
iteration 700 / 1000: loss 1.9915273825850979
iteration 800 / 1000: loss 2.0040533587870257
iteration 900 / 1000: loss 1.9480758500797803
Validation accuracy:  0.282
```

Questions:

The training accuracy isn't great.

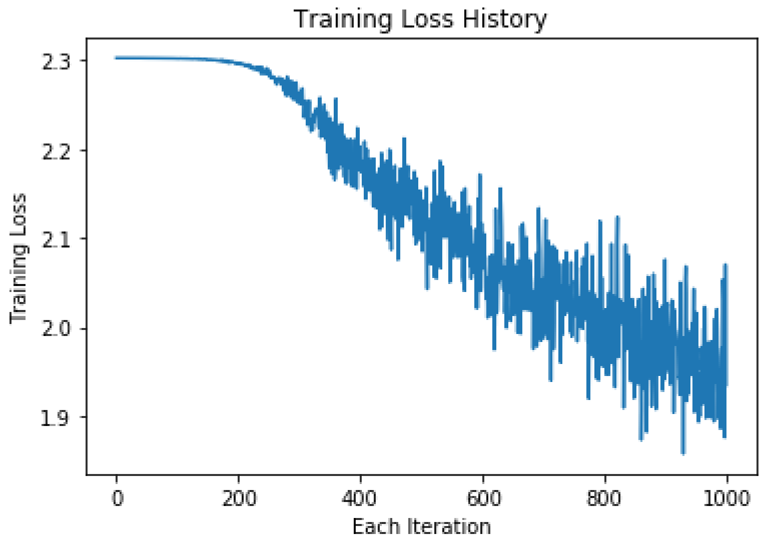
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

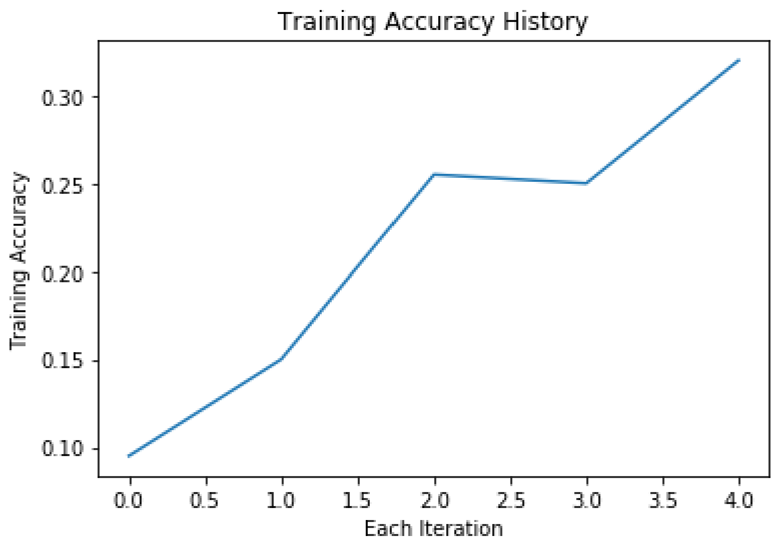
(2) How should you fix the problems you identified in (1)?

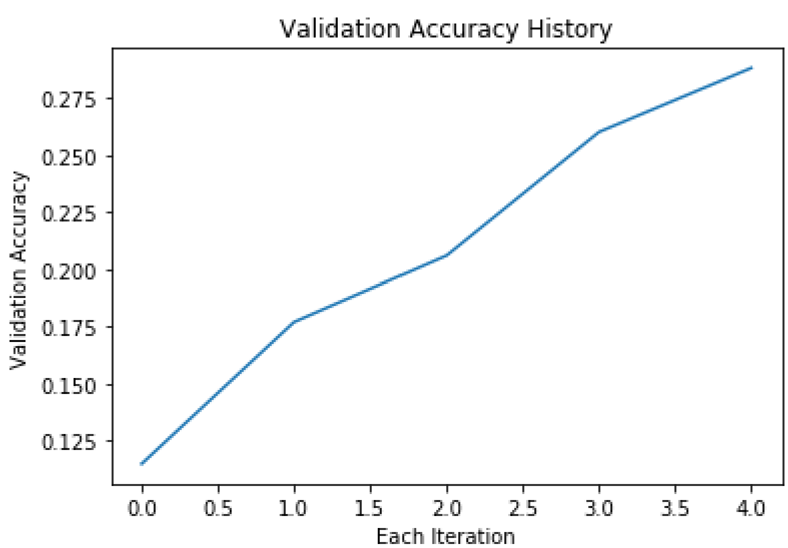
In [11]:
```
stats['train_acc_history']
```

Out[11]:
```
[0.095, 0.15, 0.255, 0.25, 0.32]
```

In [12]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies

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

plt.plot(stats['train_acc_history'])
plt.xlabel('Each Iteration')
plt.ylabel('Training Accuracy')
plt.title('Training Accuracy History')
plt.show()

plt.plot(stats['val_acc_history'])
plt.xlabel('Each Iteration')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy History')
plt.show()

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


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


Training Accuracy History
Training Accuracy
Each Iteration
0.30
0.25
0.20
0.15
0.10
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Validation Accuracy History
Validation Accuracy
Each Iteration
0.275
0.250
0.225
0.200
0.175
0.150
0.125
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Answers:

(1) As can be observed, the training loss starts zig zagging about 220 iterations in. This indicates that at about 220 iterations, our learning rate is too high. This causes the weights to overcorrect each step it takes, which resulted in the zig zag behavior seen in the graph. Additionally, the training and validation accuracies show linear behavior after 3.0. This suggests that we could train our model for more iterations until the slopes of these accuracies start to plateau.

(2) One method that many of my colleagues have used at a previous internship were adaptive learning rates. There are different ways, but Adagrad, Adam Optimization, RMSprop, momentum, etc. Additionally, the learning rate decay could be increased in order to reduce the zigzagging over each iteration. We could also increase the number of iterations to increase the validation 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.

In [13]:
```
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)!
# ================================================================ #
import time
t = time.time()
print('starting')
best_val_acc = 0.5

learning_rates = np.linspace(1e-4, 5e-3, 10)
num_iterations = [1500]
m_batch = 200
learning_rate_decays = np.linspace(0.95,0,9,5)
regs = np.linspace(0.15, 0.25, 3)
best_hyperparameters = list(range(4))

break_allloops = False

for learning_rate in learning_rates:
    if break_allloops:
        print('learning_rate')
        break
    for iters in num_iterations:
        print('num_iter')
        if break_allloops:
            break
        for decay in learning_rate_decays:
            print('decay')
            if break_allloops:
                break
            for reg in regs:
                print('reg')
                if break_allloops:
                    break
                mNet = TwoLayerNet(input_size, hidden_size, num_classes)

                stats = mNet.train(X_train, y_train, X_val, y_val, num_iters = 
iters,
                                   batch_size=m_batch, learning_rate = learning
_rate, learning_rate_decay=decay,
                                   reg=reg, verbose = False)
                val_acc = np.amax(stats['val_acc_history'])
                epoch = np.argmax(stats['val_acc_history'])
                m_iteration = 1500

                if val_acc > best_val_acc:
```

```
                    best_net = mNet

                    best_val_acc = val_acc
                    best_hyperparameters = [learning_rate, iters, decay, reg]
                    break_allloops = True
                    print(best_hyperparameters)
                    print(best_val_acc)

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

#Hyperparameters calculated are: [0.0022777777777777774, 1500, 0.8312499999999
999, 0.15]
#With validation accuracy 0.503
```

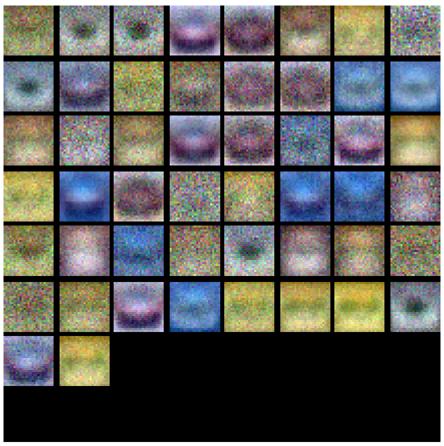
```
starting
num_iter
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
num_iter
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
```

```
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
num_iter
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
num_iter
decay
```

```
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
num_iter
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
```

```
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
decay
reg
reg
reg
num_iter
decay
reg
reg
reg
decay
reg
reg
[0.002822222222222222, 1500, 0.8312499999999999, 0.2]
0.504
reg
decay
learning_rate
Validation accuracy:  0.492
```

In [14]:
```
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 weights in the suboptimal network look very similar. There aren't many significant color changes within the suboptimal network, and the shapes of the weights seem similar as well. The best network that my debugging code above found had more varied weights in regards to shape and color.

Evaluate on test set

In [15]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.49
```

In [ ]: