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

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

Please print out the notebook entirely when completed.

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

In [5]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

Toy example

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

In [2]:
```
from nndl.neural_net import TwoLayerNet
```

In [6]:
```
# 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 [7]:
```
## 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:
3.381231248461569e-08
```

Forward pass loss

In [8]:
```
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 [11]:
```
from utils.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward
# 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], verb
    print('{} max relative error: {}'.format(param_name, rel_error(param_gra
```

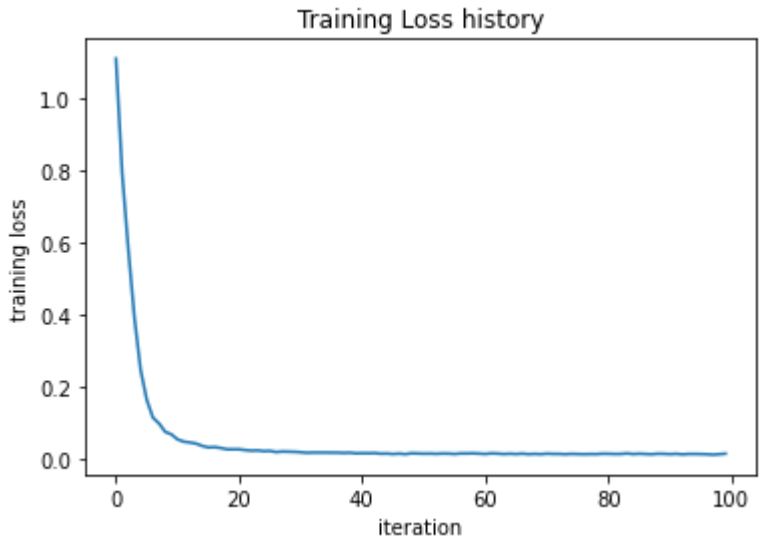
```
W1 max relative error: 1.2832874456864775e-09
b1 max relative error: 3.1726806716844575e-09
W2 max relative error: 2.9632227682005116e-10
b2 max relative error: 1.2482660547101085e-09
```

Training the network

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

In [12]:
```
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:  0.014497864587765957
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [6]:

```
from utils.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000)
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'dataset\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 [15]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

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

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

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

```
iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.251825904316413
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.9901688623083942
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

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

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

In [21]:
```
stats['train_acc_history']
```

Out[21]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

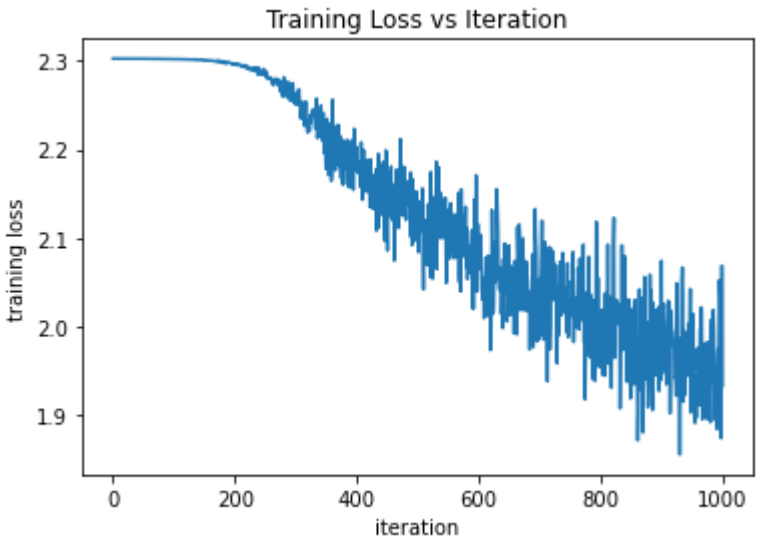
In [8]:
```
#EXPERIMENT TO SEE IF ACCURACY CAN BE IMPROVED
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=4000, batch_size=300,
            learning_rate=1e-3, 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)
```

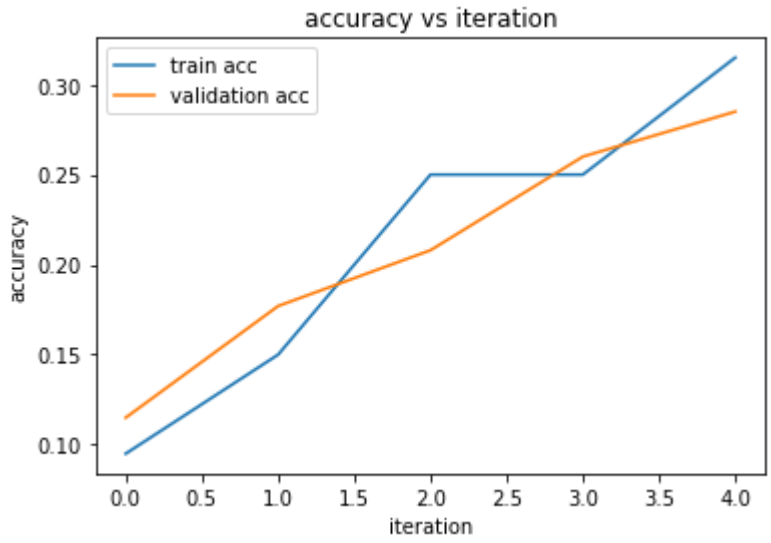
```
iteration 0 / 4000: loss 2.3028007351054924
iteration 100 / 4000: loss 1.8671721184198595
iteration 200 / 4000: loss 1.735052073342008
iteration 300 / 4000: loss 1.587012373398663
iteration 400 / 4000: loss 1.5551458368146491
iteration 500 / 4000: loss 1.572744830491382
iteration 600 / 4000: loss 1.5569265665437162
iteration 700 / 4000: loss 1.516355265953603
iteration 800 / 4000: loss 1.444393039034542
iteration 900 / 4000: loss 1.4491723548916906
iteration 1000 / 4000: loss 1.5424145271154444
iteration 1100 / 4000: loss 1.3685976268235942
iteration 1200 / 4000: loss 1.4015107602863033
iteration 1300 / 4000: loss 1.4242167509987038
iteration 1400 / 4000: loss 1.4396977624306249
iteration 1500 / 4000: loss 1.4379210168499643
iteration 1600 / 4000: loss 1.4099537449443924
iteration 1700 / 4000: loss 1.3280501659227353
iteration 1800 / 4000: loss 1.4715080321395446
iteration 1900 / 4000: loss 1.3520156785903574
iteration 2000 / 4000: loss 1.464624787212279
iteration 2100 / 4000: loss 1.4232469178947993
iteration 2200 / 4000: loss 1.4221963080109268
iteration 2300 / 4000: loss 1.4088913206433655
iteration 2400 / 4000: loss 1.2836177125029784
iteration 2500 / 4000: loss 1.3359247207809524
iteration 2600 / 4000: loss 1.3921218742224277
iteration 2700 / 4000: loss 1.349099685098755
iteration 2800 / 4000: loss 1.3981975882536557
iteration 2900 / 4000: loss 1.3972649069961904
iteration 3000 / 4000: loss 1.2974790933323546
iteration 3100 / 4000: loss 1.3364343953961237
iteration 3200 / 4000: loss 1.3796549823141266
iteration 3300 / 4000: loss 1.348516916662763
iteration 3400 / 4000: loss 1.2696567716283156
iteration 3500 / 4000: loss 1.2787863728490556
iteration 3600 / 4000: loss 1.3769728934640058
iteration 3700 / 4000: loss 1.3975589168530587
iteration 3800 / 4000: loss 1.2330940850486645
```

```
iteration 3900 / 4000: loss 1.2913398672506897
Validation accuracy:  0.514
```

In [18]:
```
# ================================================================ #
# 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()
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss vs Iteration')

plt.figure()
plt.plot(stats['train_acc_history'],label='train acc')
plt.plot(stats['val_acc_history'],label='validation acc')
plt.xlabel('iteration')
plt.ylabel('accuracy')
plt.title('accuracy vs iteration')
plt.legend()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

Out[18]: <matplotlib.legend.Legend at 0x268a3959730>





Answers:

(1) By examining the progression of the training and validation accuracies, we can determine the most obvious reason for the low accuracy as the small number of iterations the model is trained. Since both training and validation accuracies are in an increasing trend, allowing the model to run for more iterations will improve performance. After some time, we should observe that the training accuracy is increasing even if the validation accuracy does not improve. This implies overfitting and training must be stopped before this point. Furthermore, examination of training loss reveals that there is little improvement in the first 200 iterations. This implies a low learning rate. Increasing the learning rate should allow the model to improve faster and thus increase accuracy. Finally, towards the end we see a large fluctuation in loss. This can imply that because our chosen batch size is too small, we get noisy estimates of the gradient which causes the oscillations in the learning curve. Increasing the batch size should lead to a smoother loss curve, which can lead to a faster decrease in loss and increase in accuracy.

(2) I would increase the number of iterations, increase the learning rate and increase the batch size. In fact, on 2 cells above it has been shown that doing so increases the validation accuracy to 51.4%. Further optimization such as changing the regularization coefficient can also be done to improve performance.

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 [31]:

```
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)
# ================================================================ #
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
learningRates = [1e-4,1e-3,1e-2,1e-1]
regularizations = [0.3,0.2,0.1,0.01,0.001]
iterations = [10000,20000,30000]
#iterations=[10]
best_net = []
best_val_acc=0
for lr in learningRates:
    for reg in regularizations:
        for it in iterations:
            net = TwoLayerNet(input_size, hidden_size, num_classes)
            stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=it, batch_size=200,
                learning_rate=lr, learning_rate_decay= 0.95,
                reg=reg, verbose=False)
            print("Training complete")
            val_acc = (net.predict(X_val) == y_val).mean()
            print("When lr=%f, reg=%f, noIt = %d, validation accuracy=%f"%(l
            if val_acc > best_val_acc:
                best_net=net
                best_val_acc=val_acc
print("SEARCH COMPLETE")
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Training complete
When lr=0.100000, reg=0.100000, noIt = 10000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.100000, noIt = 20000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.100000, noIt = 30000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.010000, noIt = 10000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.010000, noIt = 20000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.010000, noIt = 30000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.001000, noIt = 10000, validation accuracy=0.087000
```

```
When lr=0.100000, reg=0.001000, noIt = 10000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.001000, noIt = 20000, validation accuracy=0.087000
Training complete
When lr=0.100000, reg=0.001000, noIt = 30000, validation accuracy=0.087000
SEARCH COMPLETE
Validation accuracy:  0.528
```

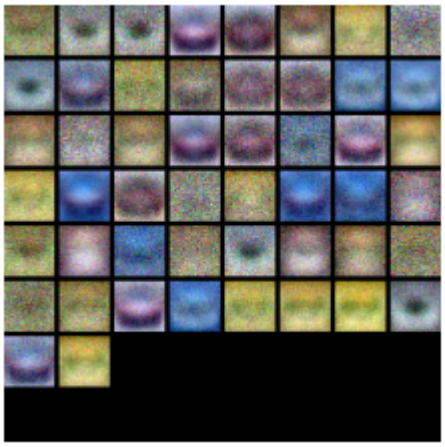
In [32]:

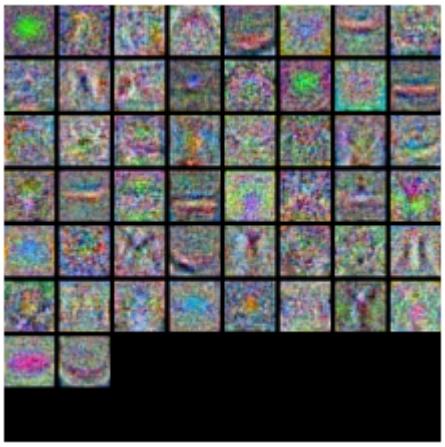
```
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.528
```

In [33]:

```
from utils.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```





Question:

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

Answer:

(1)The weights of the subopimal net appear to be noisy and without any distinguishable characteristics. It is not easy to understand the features the weights are looking at and they all look averaged in terms of shape. On the other hand, the weights of the best net offer a clear shape which implies that they learned specific features to look at. The weights are discernible from each other with distint shapes and it is easy to uderstand what each weight looks at in an image.

Evaluate on test set

In [34]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.51
```