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 [1]:

```
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 [2]:

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

In [3]:

```
# 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 [4]:

```python
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231233889892e-08
```

Forward pass loss

In [5]:

```
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 [6]:

```
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)

# 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])))
```

```
W1 max relative error: 1.2832823337649917e-09
W2 max relative error: 2.9632227682005116e-10
b1 max relative error: 3.172680092703762e-09
b2 max relative error: 1.248270530283678e-09
```

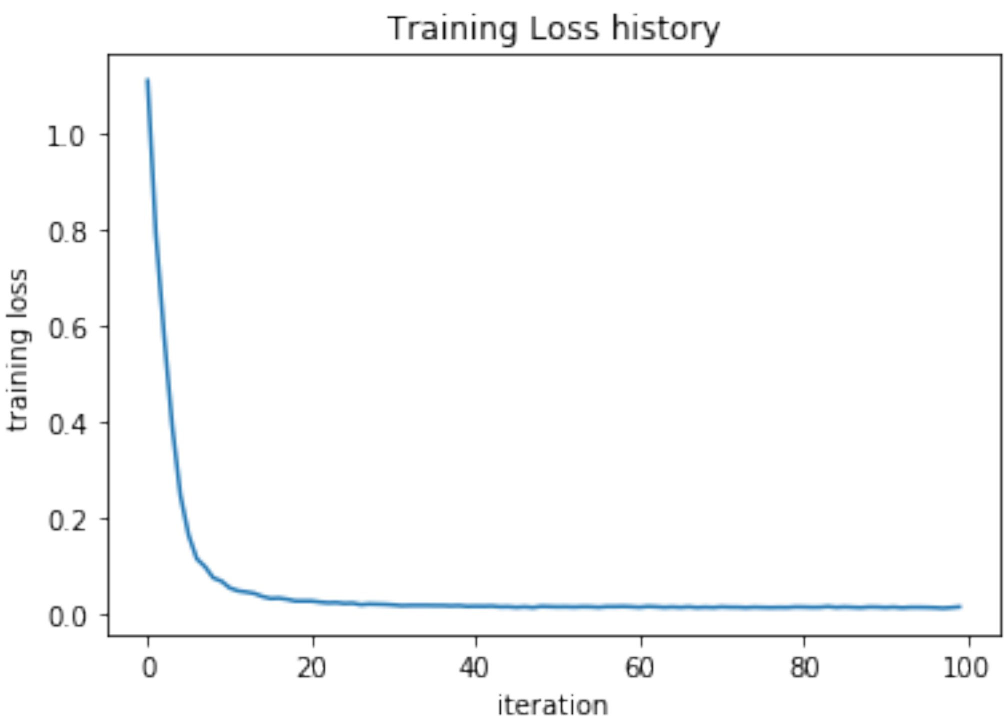
Training the network

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

In [7]:

```
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:  0.014497864587765886
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [8]:

```
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 = '/Users/egecetintas/Desktop/UCLA/c247/hw2/cifar-10-batches-p
y' # You need to update this line
    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 [9]:

```
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 2.2518259043164135
iteration 400 / 1000: loss 2.1889952350467756
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.0028276401246856
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 [10]:

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

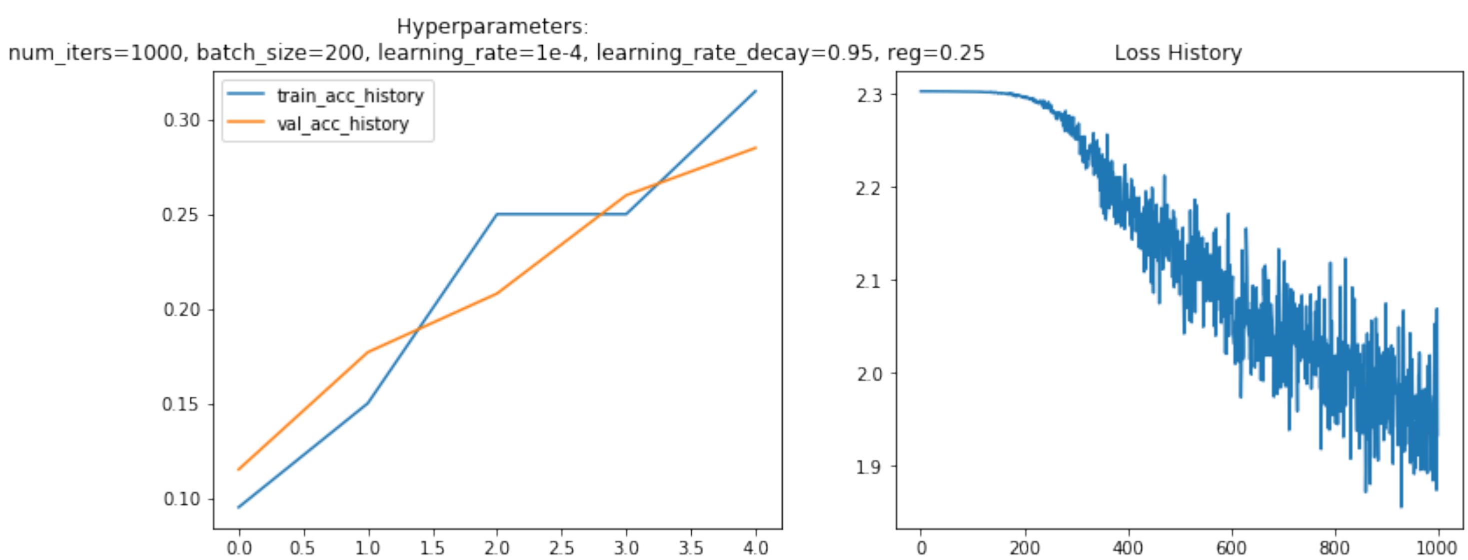
Out[10]:

```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

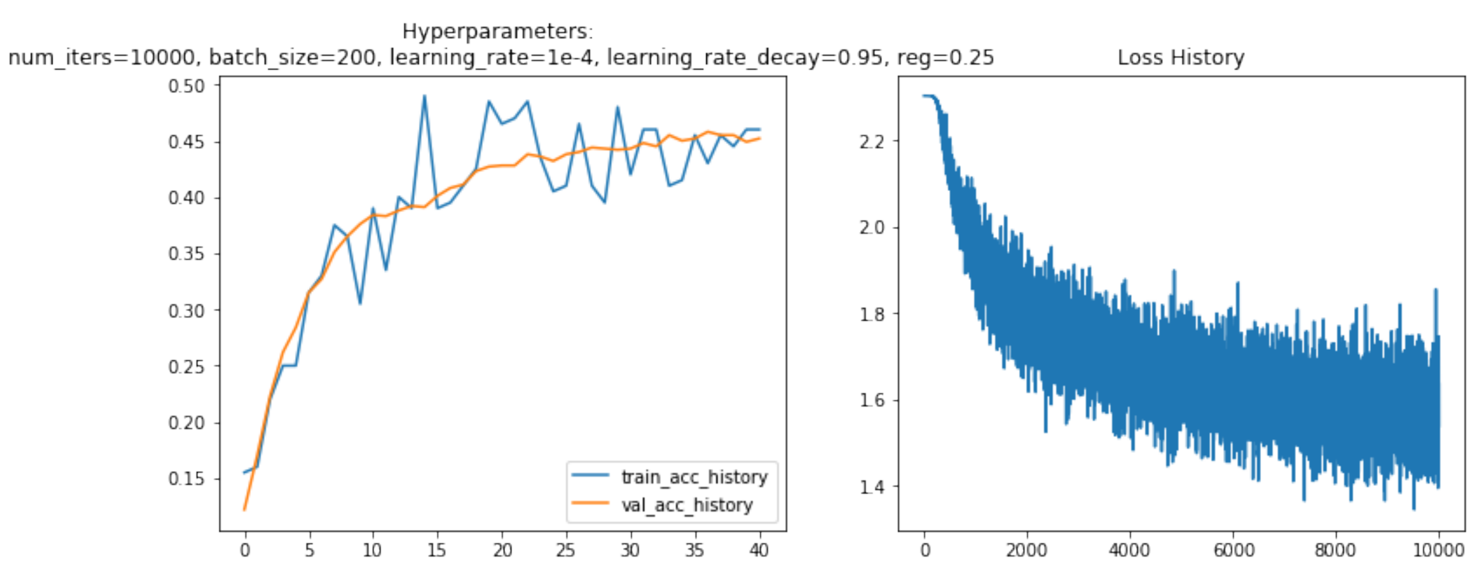
In [11]:

```
# ================================================================ #

# 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
fig = plt.figure(figsize=(2*6.4, 4.8))
plt.subplot(121)
plt.plot(stats['train_acc_history'],label='train_acc_history')
plt.plot(stats['val_acc_history'],label='val_acc_history')
plt.title('Hyperparameters: \nnum_iters=1000, batch_size=200, learning_rate=1e
-4, learning_rate_decay=0.95, reg=0.25')
plt.legend()
plt.subplot(122)
plt.title('Loss History')
plt.plot(stats['loss_history'])
plt.show()

#Trying different Number of Iteration
#1
net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=10000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=False)

val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
fig = plt.figure(figsize=(2*6.4, 4.8))
plt.subplot(121)
plt.plot(stats['train_acc_history'],label='train_acc_history')
plt.plot(stats['val_acc_history'],label='val_acc_history')
plt.title('Hyperparameters: \nnum_iters=10000, batch_size=200, learning_rate=1
e-4, learning_rate_decay=0.95, reg=0.25')
plt.legend()
plt.subplot(122)
plt.title('Loss History')
plt.plot(stats['loss_history'])
plt.show()

#2
net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=10000, batch_size=200,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=0.25, verbose=False)
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
fig = plt.figure(figsize=(2*6.4, 4.8))
plt.subplot(121)
plt.plot(stats['train_acc_history'],label='train_acc_history')
plt.plot(stats['val_acc_history'],label='val_acc_history')
plt.title('Hyperparameters: \nnum_iters=10000, batch_size=200, learning_rate=1
e-3, learning_rate_decay=0.95, reg=0.25')
plt.legend()
plt.subplot(122)
```

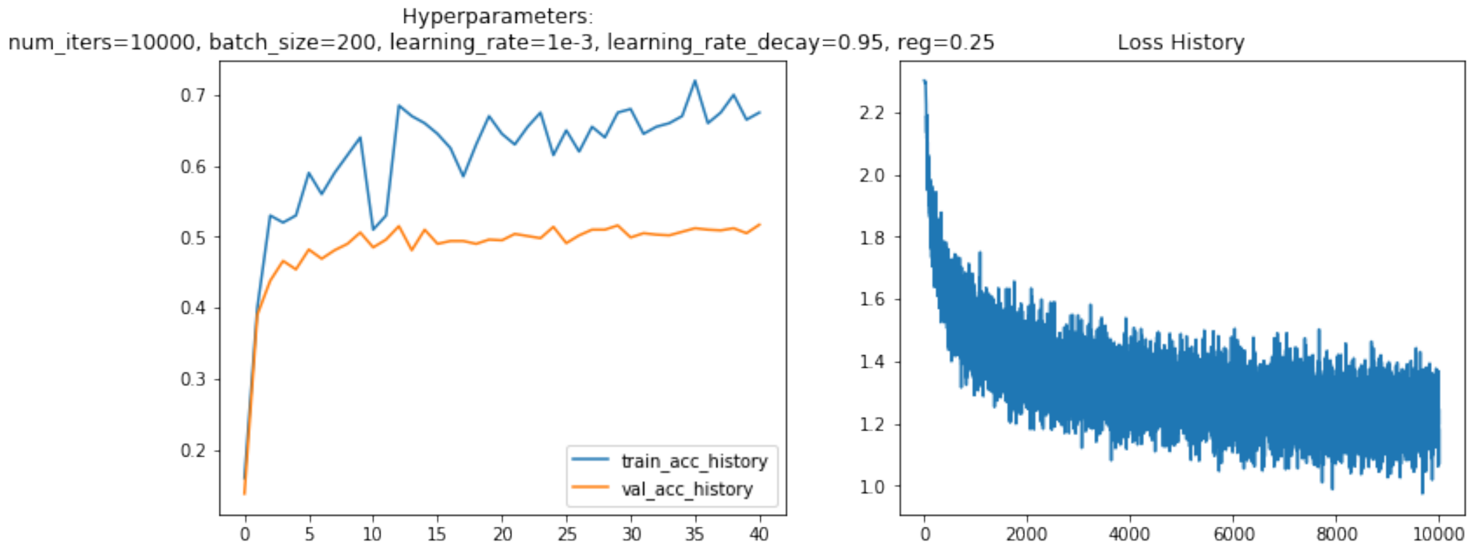
```python
plt.title('Loss History')
plt.plot(stats['loss_history'])
plt.show()

#3
net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=10000, batch_size=200,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=0.05, verbose=False)
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
fig = plt.figure(figsize=(2*6.4, 4.8))
plt.subplot(121)
plt.plot(stats['train_acc_history'],label='train_acc_history')
plt.plot(stats['val_acc_history'],label='val_acc_history')
plt.title('Hyperparameters: \nnum_iters=10000, batch_size=200, learning_rate=1
e-3, learning_rate_decay=0.95, reg=0.05')
plt.legend()
plt.subplot(122)
plt.title('Loss History')
plt.plot(stats['loss_history'])
plt.show()

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



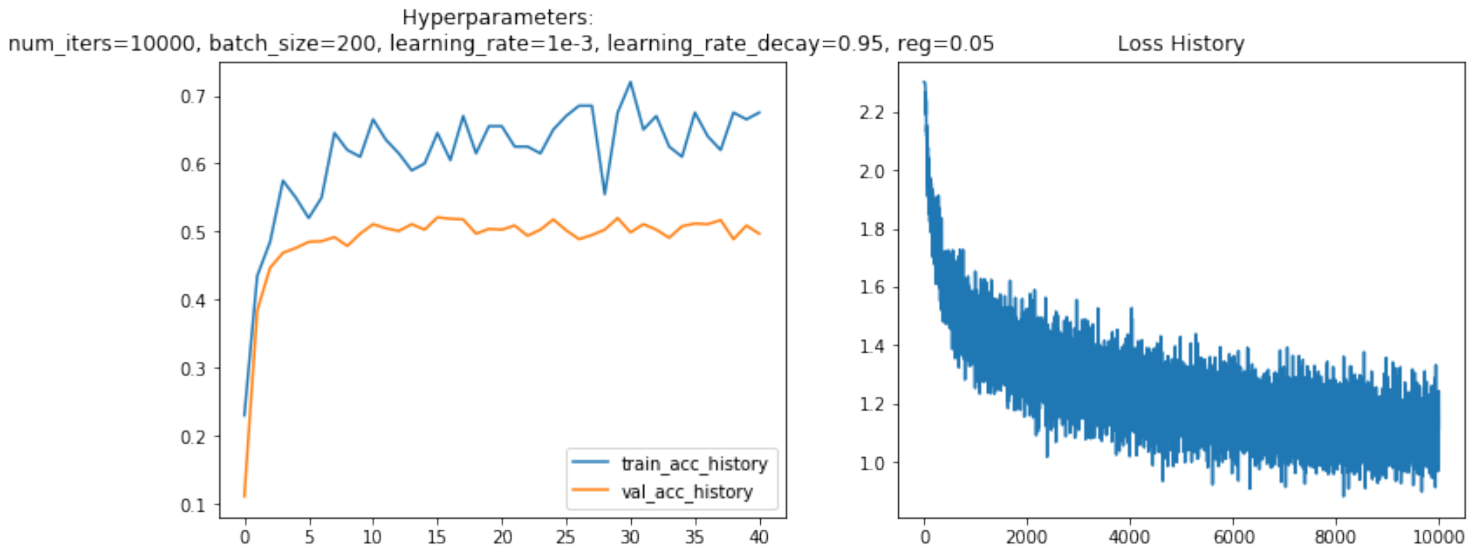

```
Validation accuracy:  0.45
```



```
Validation accuracy:  0.514
```




```
Validation accuracy:  0.499
```



Answers:

(1) The plots of loss and accuracy history above suggests that we don't achieve adequate fitting of the training data yet. Training accuracy and validation accuracy follows a similar trend -both increasing linearly- and it would suggest that there is still room for more learning. If we have a look at the loss history, we can observe that the decrease trend is still pretty linear for 1000 iteration suggesting that SGD does not find a local minima yet.

(2) In order to fix the problem, we can increase the number of iterations to force the model to learn the training data even more. Alternatively, we can play with some of the hyperparameters such as learning rate or regularization strength to optimize for a specific number of iterations.

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 [12]:

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

# ================================================================ #
# YOUR CODE HERE:
```

```
# 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

optimLength = 200

val_acc_list = []
learning_rates = []
batch_list = []
iter_list = []
reg_list = []

start_reg = 0.15
end_reg = 0.5
start_iter = 100
end_iter = 100
start_lr = -7
end_lr = -3.5
start_batch = 100
end_batch = 100
net = TwoLayerNet(input_size, hidden_size, num_classes)
for i in range(optimLength):
    #Generate Random Hyperparameters
    learning_rate_random = 10 ** np.random.uniform(start_lr, end_lr,1)
    num_iters_random = np.random.uniform(start_iter, end_iter,1)
    reg_random = np.random.uniform(start_reg,end_reg,1)
    batch_random = int(np.random.uniform(start_batch,end_batch,1))
    learning_rates.append(learning_rate_random)
    reg_list.append(reg_random)
    iter_list.append(num_iters_random)
    batch_list.append(batch_random)

    stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=num_iters_random, batch_size=batch_random,
            learning_rate=learning_rate_random, learning_rate_decay=0.95,
            reg=reg_random, verbose=False)
    val_acc = (net.predict(X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)
    val_acc_list.append(val_acc)

    hyperparamers = sorted(zip(val_acc_list,learning_rates,iter_list,reg_list,
batch_list))

    k = 10 # Intervals for Random Hyperparameters are updated every k training
.
    if (i+1) % k == 0:
        #Update Random Hyperparameters' Generation Interval
```

```
        print('\n')

        print('Current Epoch: ', str((i+1)/k))
        print('max _val_acc:',max(hyperparamers[-k:])[0])
        print('learning_rate:',max(hyperparamers[-k:])[1])
        print('iter_num:',max(hyperparamers[-k:])[2])
        print('reg_num:',max(hyperparamers[-k:])[3])
        print('batch_num:',max(hyperparamers[-k:])[4])
        print('Hyperparameter limits are updated for the next epoch!')
        print('\n')
        start_lr = np.log10(max(hyperparamers[-k:])[1]) - 0.75*0.95
        end_lr = np.log10(max(hyperparamers[-k:])[1]) + 0.75*0.95
        start_reg = max(hyperparamers[-k:])[3] - 0.05*0.95
        end_reg = max(hyperparamers[-k:])[3] + 0.1*0.95


stats = net.train(X_train, y_train, X_val, y_val,
        num_iters=100, batch_size=100,
        learning_rate=max(hyperparamers[-5:])[1][0], learning_rate_decay=0.95,
        reg=max(hyperparamers[-5:])[3][0], verbose=False)
best_net = net


pass
#Plot the val_acc history for every set of hyperparameters
plt.plot(val_acc_list)
plt.xlabel('Index of Different Hyperparameters')
plt.ylabel('Validation Loss')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.17
Validation accuracy:  0.198
Validation accuracy:  0.196
Validation accuracy:  0.196
Validation accuracy:  0.196
Validation accuracy:  0.195
Validation accuracy:  0.195
Validation accuracy:  0.195
Validation accuracy:  0.194
Validation accuracy:  0.195


Current Epoch:  1.0
max _val_acc: 0.198
learning_rate: [0.00012854]
iter_num: [100.]
reg_num: [0.30677967]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.175
Validation accuracy:  0.222
```

```
Validation accuracy:  0.247
Validation accuracy:  0.258
Validation accuracy:  0.301
Validation accuracy:  0.32
Validation accuracy:  0.334
Validation accuracy:  0.338
Validation accuracy:  0.366
Validation accuracy:  0.368


Current Epoch:  2.0
max _val_acc: 0.368
learning_rate: [3.02470654e-05]
iter_num: [100.]
reg_num: [0.31283904]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.366
Validation accuracy:  0.363
Validation accuracy:  0.364
Validation accuracy:  0.366
Validation accuracy:  0.371
Validation accuracy:  0.376
Validation accuracy:  0.373
Validation accuracy:  0.372
Validation accuracy:  0.385
Validation accuracy:  0.382


Current Epoch:  3.0
max _val_acc: 0.385
learning_rate: [7.13372609e-05]
iter_num: [100.]
reg_num: [0.33159639]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.381
Validation accuracy:  0.392
Validation accuracy:  0.39
Validation accuracy:  0.398
Validation accuracy:  0.4
Validation accuracy:  0.404
Validation accuracy:  0.408
Validation accuracy:  0.408
Validation accuracy:  0.427
Validation accuracy:  0.424


Current Epoch:  4.0
max _val_acc: 0.427
learning_rate: [0.00033178]
iter_num: [100.]
reg_num: [0.32871068]
```

```
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.433
Validation accuracy:  0.421
Validation accuracy:  0.44
Validation accuracy:  0.427
Validation accuracy:  0.465
Validation accuracy:  0.452
Validation accuracy:  0.458
Validation accuracy:  0.414
Validation accuracy:  0.455
Validation accuracy:  0.45


Current Epoch:  5.0
max _val_acc: 0.465
learning_rate: [0.00018463]
iter_num: [100.]
reg_num: [0.3408659]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.461
Validation accuracy:  0.46
Validation accuracy:  0.469
Validation accuracy:  0.474
Validation accuracy:  0.477
Validation accuracy:  0.465
Validation accuracy:  0.456
Validation accuracy:  0.482
Validation accuracy:  0.482
Validation accuracy:  0.474


Current Epoch:  6.0
max _val_acc: 0.482
learning_rate: [0.00015447]
iter_num: [100.]
reg_num: [0.43207708]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.494
Validation accuracy:  0.492
Validation accuracy:  0.49
Validation accuracy:  0.5
Validation accuracy:  0.477
Validation accuracy:  0.494
Validation accuracy:  0.501
Validation accuracy:  0.499
Validation accuracy:  0.496
Validation accuracy:  0.497
```

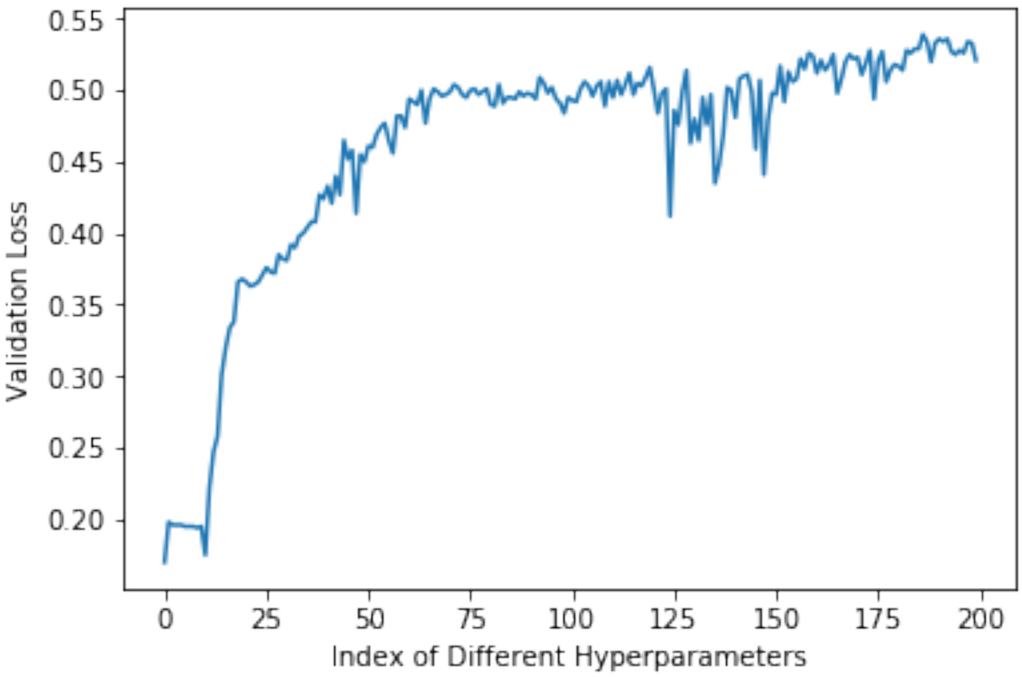
```
Current Epoch:  7.0
max _val_acc: 0.501
learning_rate: [5.57046218e-05]
iter_num: [100.]
reg_num: [0.42690496]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.499
Validation accuracy:  0.504
Validation accuracy:  0.502
Validation accuracy:  0.497
Validation accuracy:  0.495
Validation accuracy:  0.5
Validation accuracy:  0.501
Validation accuracy:  0.497
Validation accuracy:  0.499
Validation accuracy:  0.501


Current Epoch:  8.0
max _val_acc: 0.504
learning_rate: [4.81385873e-05]
iter_num: [100.]
reg_num: [0.49469929]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.49
Validation accuracy:  0.489
Validation accuracy:  0.504
Validation accuracy:  0.491
Validation accuracy:  0.495
Validation accuracy:  0.495
Validation accuracy:  0.494
Validation accuracy:  0.499
Validation accuracy:  0.496
Validation accuracy:  0.498


Current Epoch:  9.0
max _val_acc: 0.504
learning_rate: [5.35478816e-05]
iter_num: [100.]
reg_num: [0.48770629]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.497
Validation accuracy:  0.494
Validation accuracy:  0.509
Validation accuracy:  0.505
Validation accuracy:  0.498
```

```
Validation accuracy:  0.502
Validation accuracy:  0.494
Validation accuracy:  0.491
Validation accuracy:  0.484
Validation accuracy:  0.495


Current Epoch:  10.0
max _val_acc: 0.509
learning_rate: [0.00010335]
iter_num: [100.]
reg_num: [0.479229]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.493
Validation accuracy:  0.492
Validation accuracy:  0.501
Validation accuracy:  0.506
Validation accuracy:  0.503
Validation accuracy:  0.496
Validation accuracy:  0.503
Validation accuracy:  0.506
Validation accuracy:  0.489
Validation accuracy:  0.506


Current Epoch:  11.0
max _val_acc: 0.509
learning_rate: [0.00010335]
iter_num: [100.]
reg_num: [0.479229]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.495
Validation accuracy:  0.507
Validation accuracy:  0.497
Validation accuracy:  0.504
Validation accuracy:  0.512
Validation accuracy:  0.497
Validation accuracy:  0.505
Validation accuracy:  0.503
Validation accuracy:  0.509
Validation accuracy:  0.516


Current Epoch:  12.0
max _val_acc: 0.516
learning_rate: [0.00034629]
iter_num: [100.]
reg_num: [0.53231348]
batch_num: 100
Hyperparameter limits are updated for the next epoch!
```

```
Validation accuracy:  0.501
Validation accuracy:  0.484
Validation accuracy:  0.498
Validation accuracy:  0.501
Validation accuracy:  0.412
Validation accuracy:  0.486
Validation accuracy:  0.476
Validation accuracy:  0.499
Validation accuracy:  0.514
Validation accuracy:  0.463


Current Epoch:  13.0
max _val_acc: 0.516
learning_rate: [0.00034629]
iter_num: [100.]
reg_num: [0.53231348]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.48
Validation accuracy:  0.465
Validation accuracy:  0.495
Validation accuracy:  0.476
Validation accuracy:  0.497
Validation accuracy:  0.435
Validation accuracy:  0.448
Validation accuracy:  0.468
Validation accuracy:  0.502
Validation accuracy:  0.5


Current Epoch:  14.0
max _val_acc: 0.516
learning_rate: [0.00034629]
iter_num: [100.]
reg_num: [0.53231348]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.481
Validation accuracy:  0.508
Validation accuracy:  0.51
Validation accuracy:  0.511
Validation accuracy:  0.499
Validation accuracy:  0.459
Validation accuracy:  0.507
Validation accuracy:  0.441
Validation accuracy:  0.479
Validation accuracy:  0.498


Current Epoch:  15.0
max _val_acc: 0.516
```

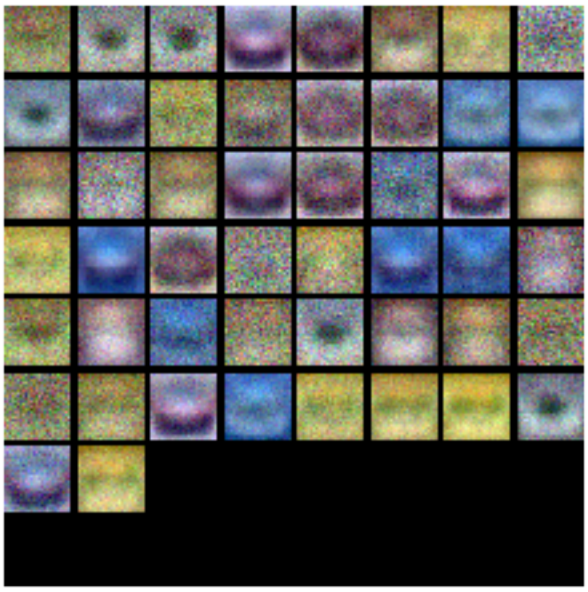
```
learning_rate: [0.00034629]
iter_num: [100.]
reg_num: [0.53231348]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.497
Validation accuracy:  0.517
Validation accuracy:  0.492
Validation accuracy:  0.513
Validation accuracy:  0.506
Validation accuracy:  0.508
Validation accuracy:  0.522
Validation accuracy:  0.515
Validation accuracy:  0.526
Validation accuracy:  0.524


Current Epoch:  16.0
max _val_acc: 0.526
learning_rate: [0.00010027]
iter_num: [100.]
reg_num: [0.5116973]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.512
Validation accuracy:  0.521
Validation accuracy:  0.514
Validation accuracy:  0.518
Validation accuracy:  0.525
Validation accuracy:  0.498
Validation accuracy:  0.508
Validation accuracy:  0.52
Validation accuracy:  0.525
Validation accuracy:  0.522


Current Epoch:  17.0
max _val_acc: 0.526
learning_rate: [0.00010027]
iter_num: [100.]
reg_num: [0.5116973]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.523
Validation accuracy:  0.511
Validation accuracy:  0.519
Validation accuracy:  0.528
Validation accuracy:  0.494
Validation accuracy:  0.52
Validation accuracy:  0.527
Validation accuracy:  0.506
```

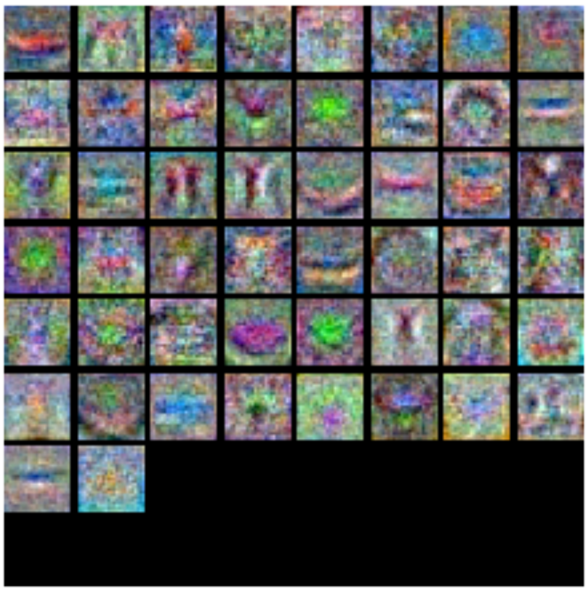
```
Validation accuracy:  0.514
Validation accuracy:  0.518


Current Epoch:  18.0
max _val_acc: 0.528
learning_rate: [6.5681754e-05]
iter_num: [100.]
reg_num: [0.50103815]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.517
Validation accuracy:  0.514
Validation accuracy:  0.528
Validation accuracy:  0.526
Validation accuracy:  0.529
Validation accuracy:  0.529
Validation accuracy:  0.539
Validation accuracy:  0.534
Validation accuracy:  0.52
Validation accuracy:  0.533


Current Epoch:  19.0
max _val_acc: 0.539
learning_rate: [2.11191102e-05]
iter_num: [100.]
reg_num: [0.52502032]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.536
Validation accuracy:  0.534
Validation accuracy:  0.536
Validation accuracy:  0.527
Validation accuracy:  0.525
Validation accuracy:  0.528
Validation accuracy:  0.526
Validation accuracy:  0.534
Validation accuracy:  0.533
Validation accuracy:  0.521


Current Epoch:  20.0
max _val_acc: 0.539
learning_rate: [2.11191102e-05]
iter_num: [100.]
reg_num: [0.52502032]
batch_num: 100
Hyperparameter limits are updated for the next epoch!


Validation accuracy:  0.536
```



In [13]:

```
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 of the best net are sharper than suboptimal net. Suboptimal net's weights are more blurred. For instance, we can reach to this conclusion by looking at the car shaped weights. The shapes are more visible in the best network's weights comparing to suboptimal net's weights.

Evaluate on test set

In [14]:

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

```
Test accuracy:  0.525
```

In [ ]:

```
```