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 [1]:

```
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. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

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]:
```
## Implement the forward pass of the neural network.
## See the loss() method in TwoLayerNet class for the same

# 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:
3.381231222787662e-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 utils.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=Fal
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num,
```

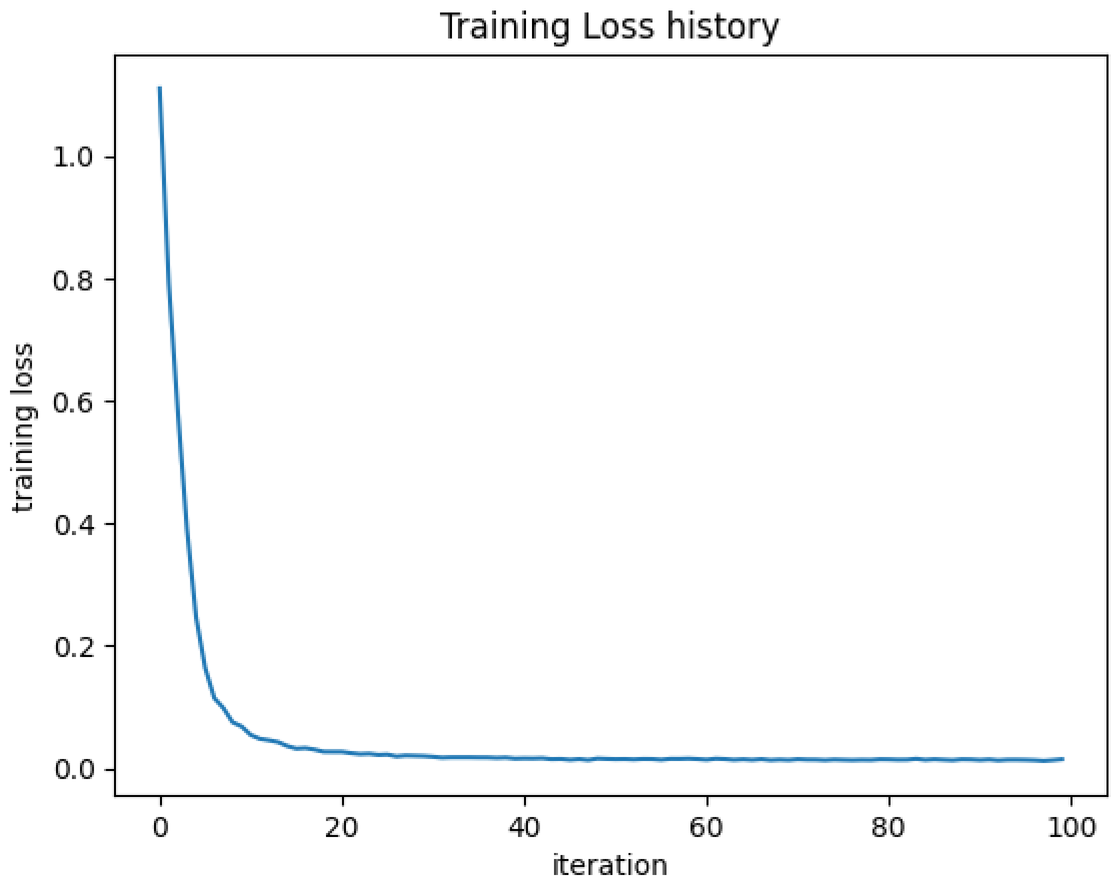
```
W2 max relative error: 2.9632227682005116e-10
b2 max relative error: 1.248270530283678e-09
W1 max relative error: 1.2832823337649917e-09
b1 max relative error: 3.172680092703762e-09
```

Training the network

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

In [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 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 = 'C:/Users/jessi/OneDrive/Desktop/classes/c147/HW2_code/cifar-10-b
    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 2.188995235046776
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.

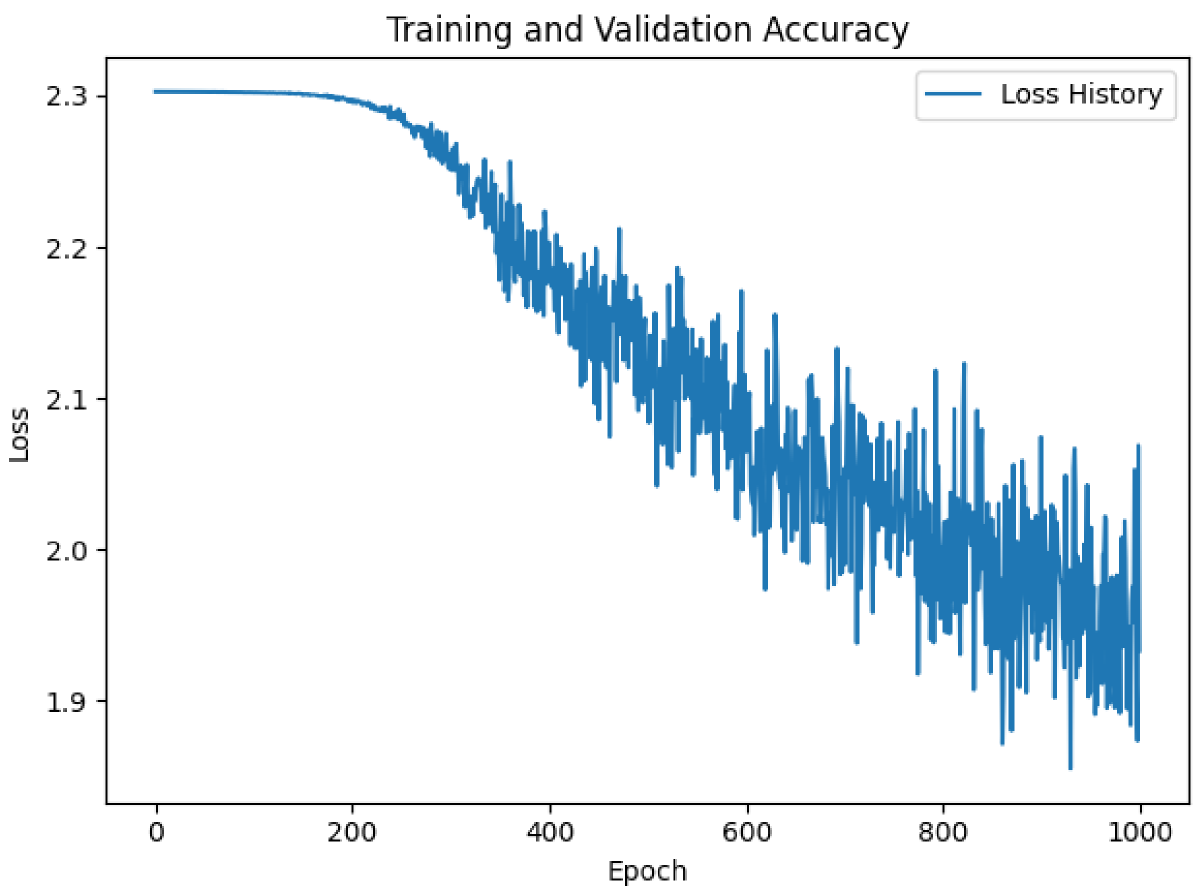
(2) How should you fix the problems you identified in (1)?

In [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
plt.figure()
plt.plot(stats['train_acc_history'], label='Training Acc')
plt.plot(stats['val_acc_history'], label='Validation Acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Loss')
plt.legend()

# Plotting the accuracies
plt.figure()
plt.plot(stats['loss_history'], label='Loss History')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.tight_layout()
plt.show()


pass
```

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



Answers:

(1) Some reasons why the validation accuracy is low might be that the learning rate needs to be ajusted, or since the training accuracy isn't great, we might need to increase the complexity of the model. (2) We can try to improve the model by trying out different learning rates, regularization coefficients, and learning rate decay coefficients.

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]:
```
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.
```

```python
#
#   Note, you need to use the same network structure (keep hidden_size = 50)!
# ================================================================ #
val_accs = []
lr = [0.1, 0.01, 0.001, 1e-4, 1e-5, 1e-6, 1e-7, 1e-7, 1e-8, 1e-9]

for r in range(len(lr)):
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=1000, batch_size=200,
                learning_rate=lr[r], learning_rate_decay=0.95,
                reg=0.25, verbose=False)

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


best_acc = np.max(val_accs)
best_lr = lr[np.argmax(val_accs)]
print("best accuracy was: ", str(np.max(val_accs)), "with learning rate =", str(lr[

pass

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

```
c:\Users\jessi\OneDrive\Desktop\classes\c147\HW3_code\HW3_code\nndl\neural_net.py:11
6: RuntimeWarning: divide by zero encountered in log
  softmax_loss = -np.sum(np.log(probs[np.arange(N), y])) / N
c:\Users\jessi\OneDrive\Desktop\classes\c147\HW3_code\HW3_code\nndl\neural_net.py:11
3: RuntimeWarning: overflow encountered in subtract
  exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))
c:\Users\jessi\OneDrive\Desktop\classes\c147\HW3_code\HW3_code\nndl\neural_net.py:11
3: RuntimeWarning: invalid value encountered in subtract
  exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Validation accuracy:  0.087
Validation accuracy:  0.087
Validation accuracy:  0.458
Validation accuracy:  0.286
Validation accuracy:  0.226
Validation accuracy:  0.142
Validation accuracy:  0.127
Validation accuracy:  0.116
Validation accuracy:  0.114
Validation accuracy:  0.094
best accuracy was:  0.458 with learning rate = 0.001
```

In [78]:

```python
val_accs_reg = []
regs = [0.25, 0.1, 0.01, 0.001, 1e-4, 1e-5, 1e-6, 1e-7, 1e-7, 1e-8, 1e-9]

for r in range(len(regs)):
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
```

```
                num_iters=1000, batch_size=200,
                learning_rate=best_lr, learning_rate_decay=0.95,
                reg=regs[r], verbose=False)

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


best_acc = np.max(val_accs_reg)
best_reg = regs[np.argmax(val_accs_reg)]
print("best accuracy was: ", str(np.max(val_accs_reg)), "with reg =", str(regs[np.a
```

```
Validation accuracy:  0.478
Validation accuracy:  0.485
Validation accuracy:  0.488
Validation accuracy:  0.48
Validation accuracy:  0.473
Validation accuracy:  0.471
Validation accuracy:  0.476
Validation accuracy:  0.483
Validation accuracy:  0.464
Validation accuracy:  0.466
Validation accuracy:  0.461
best accuracy was:  0.488 with reg = 0.01
```

In [88]:

```
val_accs_decay = []
decays = [1, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5]

for r in range(len(regs)):
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=1000, batch_size=200,
                learning_rate=best_lr, learning_rate_decay=decays[r],
                reg=best_reg, verbose=False)

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


best_acc = np.max(val_accs_decay)
best_decay = decays[np.argmax(val_accs_decay)]
print("best accuracy was: ", str(np.max(val_accs_decay)), "with learning rate decay
```

```
Validation accuracy:  0.445
Validation accuracy:  0.494
Validation accuracy:  0.469
Validation accuracy:  0.47
Validation accuracy:  0.472
Validation accuracy:  0.456
Validation accuracy:  0.462
Validation accuracy:  0.437
Validation accuracy:  0.439
Validation accuracy:  0.419
Validation accuracy:  0.387
best accuracy was:  0.494 with learning rate decay = 0.95
```

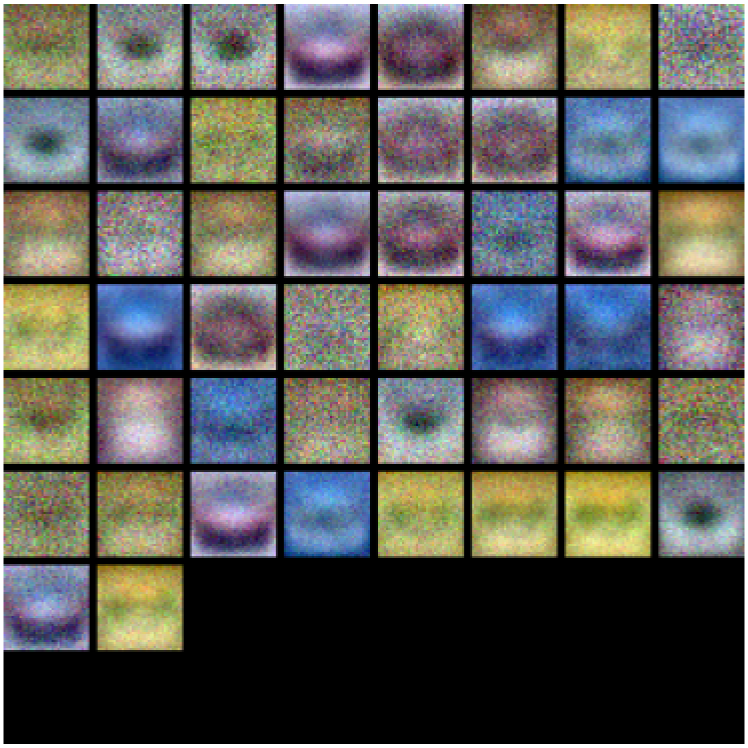
In [104…

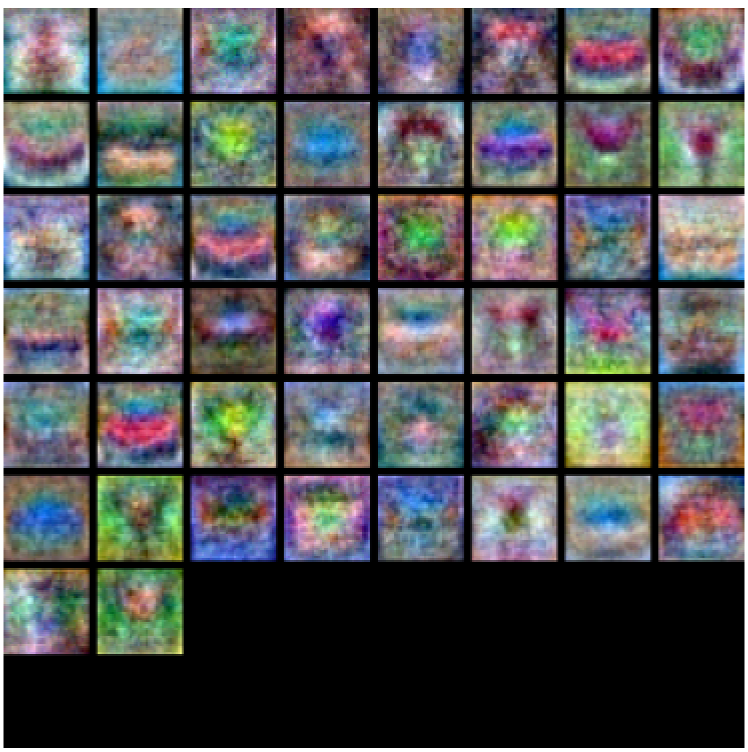
```
best_net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = best_net.train(X_train, y_train, X_val, y_val,
                num_iters=1000, batch_size=200,
                learning_rate=0.001, learning_rate_decay=0.95,
                reg=0.01, verbose=False)
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.499
```

In [55]:

```
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 in the net I arrived at look a lot closer to objects that might correspond to photos from the CIFAR-10 dataset than the suboptimal net. There are more colors and more distiguishable groups of pixels, aka less noise.

Evaluate on test set

In [110…

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

```
Test accuracy:  0.464
```