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]:
```
## 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:
3.381231204052648e-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('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

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

In [6]:
```
print(loss)
```

```
1.071696123862817
```

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 pas
s.
# 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=F
alse)
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num
, grads[param_name])))
```

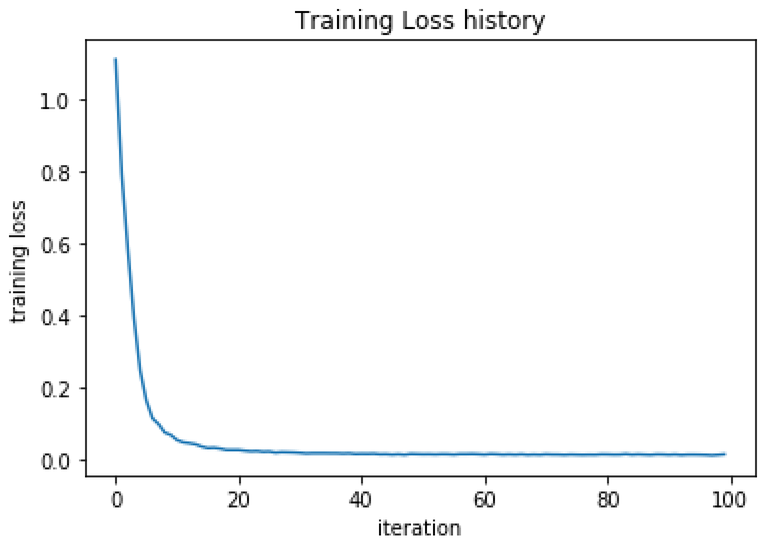
```
W2 max relative error: 2.9632233460136427e-10
b2 max relative error: 1.8392106647421603e-10
W1 max relative error: 1.283286893046317e-09
b1 max relative error: 3.1726799962069797e-09
```

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:  0.014497864587765595
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [9]:
```python
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 2.302120159207236
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.

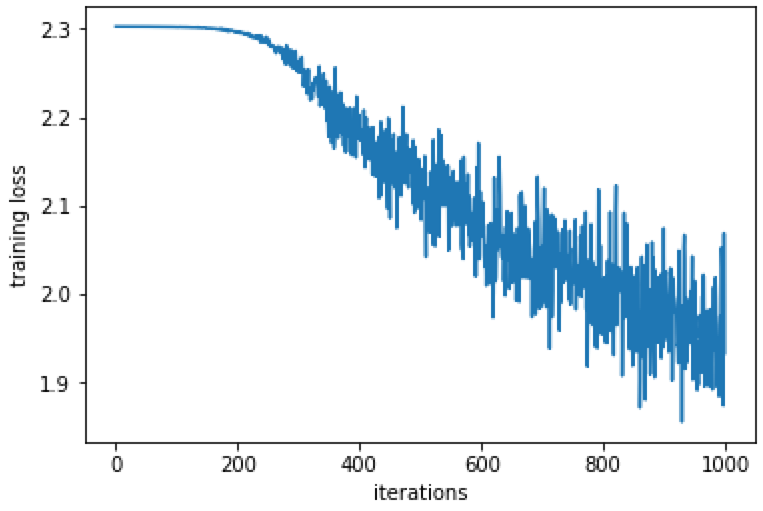
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

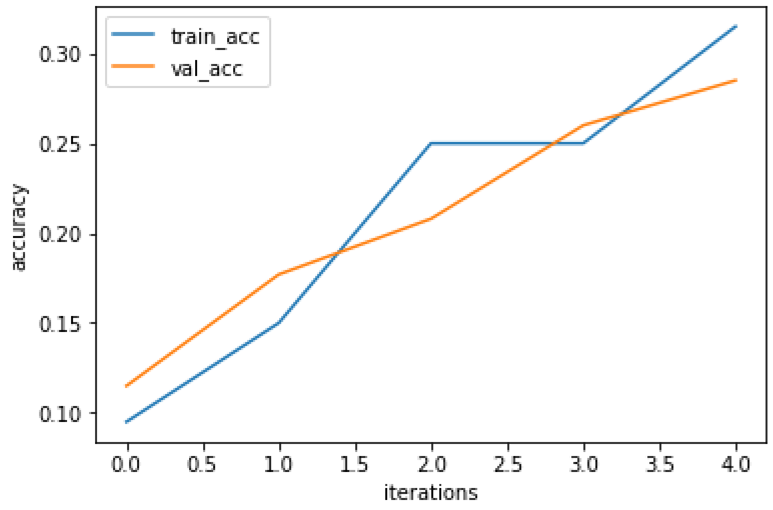
(2) How should you fix the problems you identified in (1)?

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

Out[11]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

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('iterations')
plt.ylabel('training loss')
plt.show()

train_plt = plt.plot(stats['train_acc_history'], label='train_acc')
val_plt = plt.plot(stats['val_acc_history'], label='val_acc')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(handles=[train_plt[0], val_plt[0]])
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```





Answers:

(1) Network needs more iterations; this is evidenced by the fact that the training loss hasn't flattened out yet.

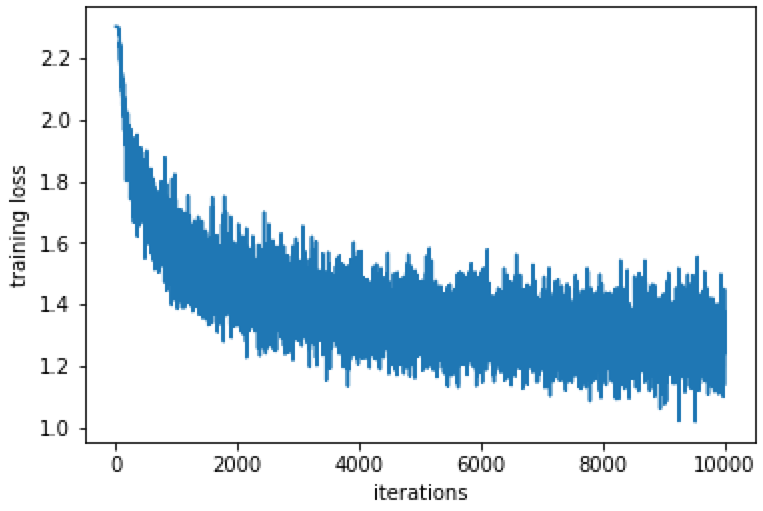
(2) Train the network on more iterations.

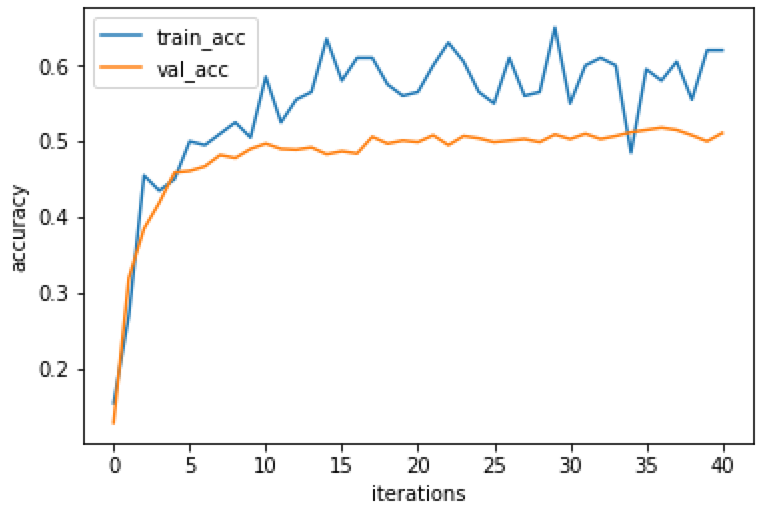
Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

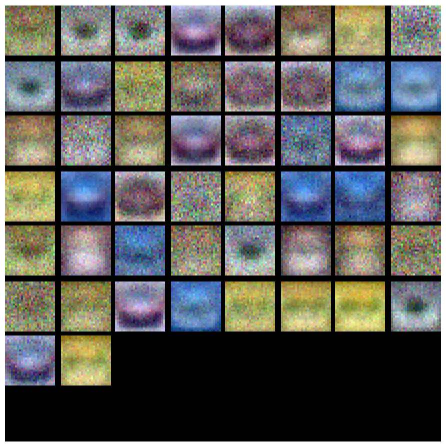
In [13]:
```
best_net = None # store the best model into this

# ================================================================ #
# 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)!
# ================================================================ #
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
# Changed num_iters from 1000 to 10000
# Changed learning_rate from 1e-4 to 5e-4
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=10000, batch_size=200,
            learning_rate=5e-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)

plt.plot(stats['loss_history'])
plt.xlabel('iterations')
plt.ylabel('training loss')
plt.show()

train_plt = plt.plot(stats['train_acc_history'], label='train_acc')
val_plt = plt.plot(stats['val_acc_history'], label='val_acc')
plt.xlabel('iterations')
plt.ylabel('accuracy')
plt.legend(handles=[train_plt[0], val_plt[0]])
plt.show()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
best_net = net
```

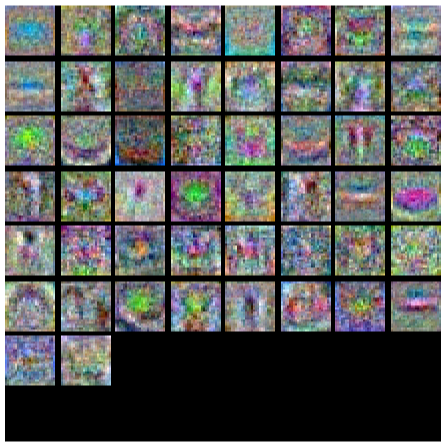
```
iteration 0 / 10000: loss 2.3027667167979295
iteration 100 / 10000: loss 2.150700404323184
iteration 200 / 10000: loss 1.879372419292919
iteration 300 / 10000: loss 1.9023182130827285
iteration 400 / 10000: loss 1.7807142164056704
iteration 500 / 10000: loss 1.8181766769644458
iteration 600 / 10000: loss 1.6414543683069336
iteration 700 / 10000: loss 1.6940316212826791
iteration 800 / 10000: loss 1.6196747665938407
iteration 900 / 10000: loss 1.6406450319379673
iteration 1000 / 10000: loss 1.5120425119723269
iteration 1100 / 10000: loss 1.494769579559985
iteration 1200 / 10000: loss 1.531886187331089
iteration 1300 / 10000: loss 1.547155005442258
iteration 1400 / 10000: loss 1.5132851154937077
iteration 1500 / 10000: loss 1.453939234243126
iteration 1600 / 10000: loss 1.461623947034721
iteration 1700 / 10000: loss 1.547346288758514
iteration 1800 / 10000: loss 1.3682624643370256
iteration 1900 / 10000: loss 1.5815310032238992
iteration 2000 / 10000: loss 1.4679435396646277
iteration 2100 / 10000: loss 1.4049537929212006
iteration 2200 / 10000: loss 1.3817505548517754
iteration 2300 / 10000: loss 1.4495089637987175
iteration 2400 / 10000: loss 1.328465196790276
iteration 2500 / 10000: loss 1.4310819970683502
iteration 2600 / 10000: loss 1.5498635334664423
iteration 2700 / 10000: loss 1.5402525253364188
iteration 2800 / 10000: loss 1.4329429930813415
iteration 2900 / 10000: loss 1.4112411804077145
iteration 3000 / 10000: loss 1.3973912564253008
iteration 3100 / 10000: loss 1.572805577184825
iteration 3200 / 10000: loss 1.359371239528938
iteration 3300 / 10000: loss 1.3811747563387333
iteration 3400 / 10000: loss 1.4160773841356475
iteration 3500 / 10000: loss 1.4195009141239936
iteration 3600 / 10000: loss 1.350825460475455
iteration 3700 / 10000: loss 1.2911371561539673
iteration 3800 / 10000: loss 1.3731407597629282
iteration 3900 / 10000: loss 1.4124538383982286
iteration 4000 / 10000: loss 1.4441994055060885
iteration 4100 / 10000: loss 1.3246583909471457
iteration 4200 / 10000: loss 1.3624120124779688
iteration 4300 / 10000: loss 1.275014816492637
iteration 4400 / 10000: loss 1.3528064446794132
iteration 4500 / 10000: loss 1.3354325618133192
iteration 4600 / 10000: loss 1.3278215295604259
iteration 4700 / 10000: loss 1.3217036343822535
iteration 4800 / 10000: loss 1.3801300810517763
iteration 4900 / 10000: loss 1.3491733661265153
iteration 5000 / 10000: loss 1.3233163127059862
iteration 5100 / 10000: loss 1.4413107772170564
iteration 5200 / 10000: loss 1.2665377397971052
iteration 5300 / 10000: loss 1.3803130359901452
iteration 5400 / 10000: loss 1.309787842974287
iteration 5500 / 10000: loss 1.2774451597439764
iteration 5600 / 10000: loss 1.3270047840876893
iteration 5700 / 10000: loss 1.3499470249137107
iteration 5800 / 10000: loss 1.5044287004932146
iteration 5900 / 10000: loss 1.5049673726137842
iteration 6000 / 10000: loss 1.255532126006217
iteration 6100 / 10000: loss 1.5777025416034964
iteration 6200 / 10000: loss 1.2319064826917787
iteration 6300 / 10000: loss 1.3651920885472617
iteration 6400 / 10000: loss 1.2136066551065452
iteration 6500 / 10000: loss 1.4709769773807257
iteration 6600 / 10000: loss 1.2491033618943153
iteration 6700 / 10000: loss 1.2282269821290899
iteration 6800 / 10000: loss 1.2192034678544237
iteration 6900 / 10000: loss 1.2626648985642135
iteration 7000 / 10000: loss 1.3302038264636649
iteration 7100 / 10000: loss 1.2818110960312208
iteration 7200 / 10000: loss 1.3066055244665762
iteration 7300 / 10000: loss 1.3552036062279216
iteration 7400 / 10000: loss 1.3200377322348247
iteration 7500 / 10000: loss 1.3203374770555931
iteration 7600 / 10000: loss 1.2074896307210108
iteration 7700 / 10000: loss 1.4101472056848172
iteration 7800 / 10000: loss 1.2377970622028498
iteration 7900 / 10000: loss 1.2509436584468023
iteration 8000 / 10000: loss 1.2519056490843126
iteration 8100 / 10000: loss 1.356874812752282
iteration 8200 / 10000: loss 1.096048928736568
iteration 8300 / 10000: loss 1.1368053760281858
iteration 8400 / 10000: loss 1.2959323614794833
iteration 8500 / 10000: loss 1.206817959804406
iteration 8600 / 10000: loss 1.3717791188775452
iteration 8700 / 10000: loss 1.3697091377560717
iteration 8800 / 10000: loss 1.3641535991107125
iteration 8900 / 10000: loss 1.2475129846109128
iteration 9000 / 10000: loss 1.1826295062685948
iteration 9100 / 10000: loss 1.2716632836494195
iteration 9200 / 10000: loss 1.234386593980497
iteration 9300 / 10000: loss 1.2175029237525152
iteration 9400 / 10000: loss 1.3239823811498188
iteration 9500 / 10000: loss 1.2078180564723775
iteration 9600 / 10000: loss 1.2710874817084576
iteration 9700 / 10000: loss 1.3382242965166542
iteration 9800 / 10000: loss 1.204562418055099
iteration 9900 / 10000: loss 1.3023757043096909
Validation accuracy:  0.508
```





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 suboptimal net looks a lot more just like averages of certain image classes, especially the car, whereas the best net is pretty abstract and grainy looking, more like certain shapes are being learned.

Evaluate on test set

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

```
Test accuracy:  0.525
```