This is the 2-layer neural network workbook for ECE 247 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:
3.381231248461569e-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 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])))
```

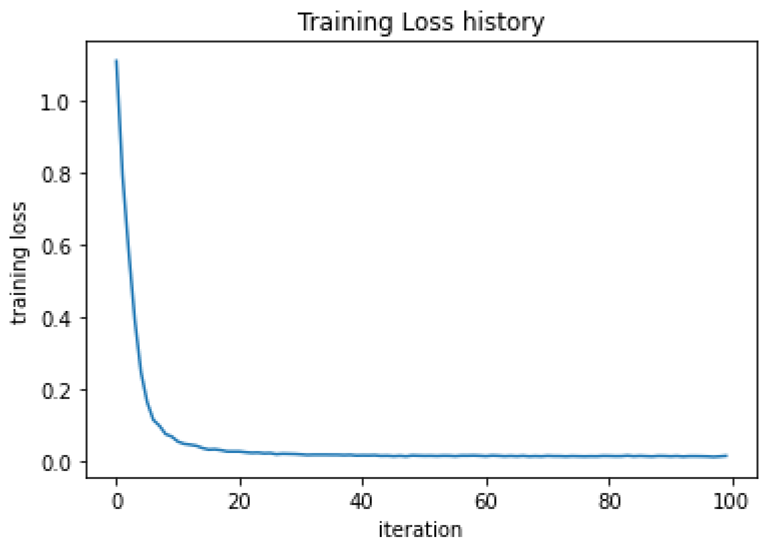
```
W1 max relative error: 1.2832892417669998e-09
W2 max relative error: 2.9632233460136427e-10
b1 max relative error: 3.172680285697327e-09
b2 max relative error: 1.2482624742512528e-09
```

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:  0.014497864587765906
```



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 = r'C:\Users\lpott\Desktop\UCLA\ECENGR247C-80\HW2\cifar-10-bat
ches-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 2.251825904316413
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

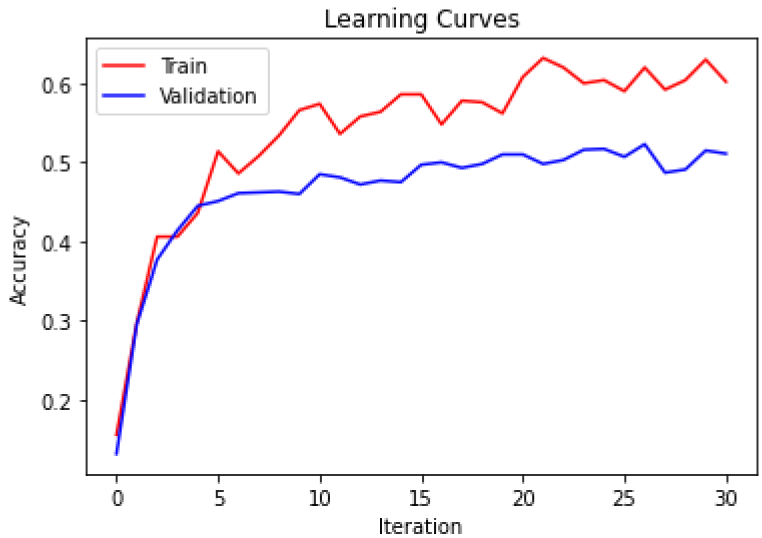
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

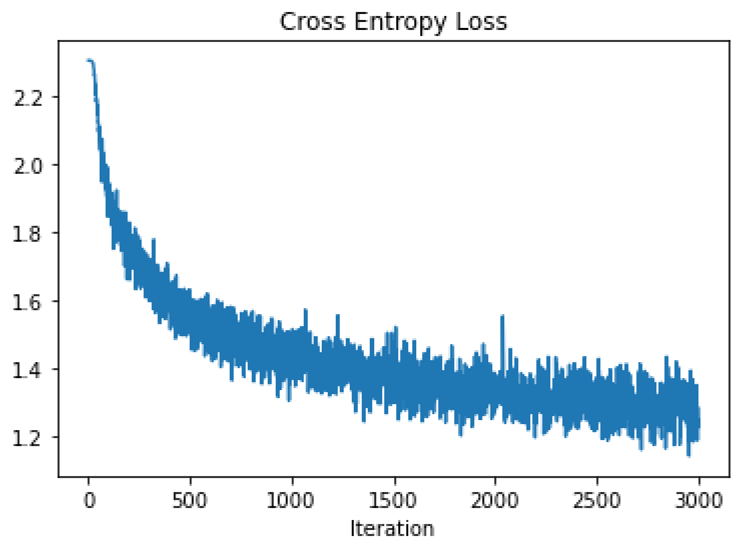
(2) How should you fix the problems you identified in (1)?

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
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
batch_size = 500
learning_rate = 1e-3
learning_rate_decay = .99
reg = .2
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=3000, batch_size=batch_size,
            learning_rate=learning_rate, learning_rate_decay=learning_rate_dec
ay,
            reg=reg, verbose=False)

plt.plot(stats['train_acc_history'],'r-')
plt.plot(stats['val_acc_history'],'b-')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Learning Curves')
plt.legend(['Train','Validation'])

plt.figure()
plt.plot(stats['loss_history'])
plt.title('Cross Entropy Loss')
plt.xlabel('Iteration')
pass
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```





Answers:

(1) I noticed the trend that the regularization weight for W1 and W2 was too high, the batch size was too small, the learning rate was too small, the learning rate decay could be too high, and increasing the number of iterations helps.

(2) I should perform a grid-hyperparameter search to identify the optimal hyperparamer values for the batch size, learning rate, learning rate decay, and regularization weight.

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

learning_rates = 1/np.logspace(3,6,num=10)
decay_rates = [.99,.95,.9,.5]
regularization = [0,0.05,.1,.2,.25,.5,.9,.99]
batch_sizes = [64,200,500]
best_acc = 0
for learning_rate in learning_rates:
    for learning_rate_decay in decay_rates:
        for reg in regularization:
            for batch_size in batch_sizes:
                net = TwoLayerNet(input_size, hidden_size, num_classes)

                # Train the network
                stats = net.train(X_train, y_train, X_val, y_val,
                            num_iters=3000, batch_size=batch_size,
                            learning_rate=learning_rate, learning_rate_decay=l
earning_rate_decay,
                            reg=reg, verbose=False)

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

                if best_acc < val_acc:
                    best_acc = val_acc
                    best_net = net
                pass

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

```
0.001 0.99 0 64
Validation accuracy:  0.455
0.001 0.99 0 200
Validation accuracy:  0.519
0.001 0.99 0 500
Validation accuracy:  0.506
0.001 0.99 0.05 64
Validation accuracy:  0.454
0.001 0.99 0.05 200
Validation accuracy:  0.502
0.001 0.99 0.05 500
Validation accuracy:  0.51
0.001 0.99 0.1 64
Validation accuracy:  0.448
0.001 0.99 0.1 200
Validation accuracy:  0.494
0.001 0.99 0.1 500
Validation accuracy:  0.515
0.001 0.99 0.2 64
Validation accuracy:  0.479
0.001 0.99 0.2 200
Validation accuracy:  0.482
0.001 0.99 0.2 500
Validation accuracy:  0.493
0.001 0.99 0.25 64
Validation accuracy:  0.457
0.001 0.99 0.25 200
Validation accuracy:  0.516
0.001 0.99 0.25 500
Validation accuracy:  0.522
0.001 0.99 0.5 64
Validation accuracy:  0.468
0.001 0.99 0.5 200
Validation accuracy:  0.48
0.001 0.99 0.5 500
Validation accuracy:  0.494
0.001 0.99 0.9 64
Validation accuracy:  0.439
0.001 0.99 0.9 200
Validation accuracy:  0.482
0.001 0.99 0.9 500
Validation accuracy:  0.515
0.001 0.99 0.99 64
Validation accuracy:  0.428
0.001 0.99 0.99 200
Validation accuracy:  0.49
0.001 0.99 0.99 500
Validation accuracy:  0.496
0.001 0.95 0 64
Validation accuracy:  0.48
0.001 0.95 0 200
Validation accuracy:  0.512
0.001 0.95 0 500
Validation accuracy:  0.497
0.001 0.95 0.05 64
Validation accuracy:  0.449
0.001 0.95 0.05 200
```

```
Validation accuracy:  0.521
0.001 0.95 0.05 500
Validation accuracy:  0.512
0.001 0.95 0.1 64
Validation accuracy:  0.472
0.001 0.95 0.1 200
Validation accuracy:  0.51
0.001 0.95 0.1 500
Validation accuracy:  0.502
0.001 0.95 0.2 64
Validation accuracy:  0.462
0.001 0.95 0.2 200
Validation accuracy:  0.513
0.001 0.95 0.2 500
Validation accuracy:  0.51
0.001 0.95 0.25 64
Validation accuracy:  0.433
0.001 0.95 0.25 200
Validation accuracy:  0.505
0.001 0.95 0.25 500
Validation accuracy:  0.508
0.001 0.95 0.5 64
Validation accuracy:  0.481
0.001 0.95 0.5 200
Validation accuracy:  0.504
0.001 0.95 0.5 500
Validation accuracy:  0.509
0.001 0.95 0.9 64
Validation accuracy:  0.431
0.001 0.95 0.9 200
Validation accuracy:  0.498
0.001 0.95 0.9 500
Validation accuracy:  0.5
0.001 0.95 0.99 64
Validation accuracy:  0.444
0.001 0.95 0.99 200
Validation accuracy:  0.489
0.001 0.95 0.99 500
Validation accuracy:  0.492
0.001 0.9 0 64
Validation accuracy:  0.457
0.001 0.9 0 200
Validation accuracy:  0.515
0.001 0.9 0 500
Validation accuracy:  0.476
0.001 0.9 0.05 64
Validation accuracy:  0.447
0.001 0.9 0.05 200
Validation accuracy:  0.511
0.001 0.9 0.05 500
Validation accuracy:  0.483
0.001 0.9 0.1 64
Validation accuracy:  0.469
0.001 0.9 0.1 200
Validation accuracy:  0.508
0.001 0.9 0.1 500
Validation accuracy:  0.476
```

```
0.001 0.9 0.2 64
Validation accuracy:  0.451
0.001 0.9 0.2 200
Validation accuracy:  0.496
0.001 0.9 0.2 500
Validation accuracy:  0.497
0.001 0.9 0.25 64
Validation accuracy:  0.473
0.001 0.9 0.25 200
Validation accuracy:  0.507
0.001 0.9 0.25 500
Validation accuracy:  0.485
0.001 0.9 0.5 64
Validation accuracy:  0.462
0.001 0.9 0.5 200
Validation accuracy:  0.506
0.001 0.9 0.5 500
Validation accuracy:  0.492
0.001 0.9 0.9 64
Validation accuracy:  0.487
0.001 0.9 0.9 200
Validation accuracy:  0.491
0.001 0.9 0.9 500
Validation accuracy:  0.482
0.001 0.9 0.99 64
Validation accuracy:  0.431
0.001 0.9 0.99 200
Validation accuracy:  0.513
0.001 0.9 0.99 500
Validation accuracy:  0.483
0.001 0.5 0 64
Validation accuracy:  0.479
0.001 0.5 0 200
Validation accuracy:  0.396
0.001 0.5 0 500
Validation accuracy:  0.289
0.001 0.5 0.05 64
Validation accuracy:  0.489
0.001 0.5 0.05 200
Validation accuracy:  0.404
0.001 0.5 0.05 500
Validation accuracy:  0.291
0.001 0.5 0.1 64
Validation accuracy:  0.458
0.001 0.5 0.1 200
Validation accuracy:  0.401
0.001 0.5 0.1 500
Validation accuracy:  0.29
0.001 0.5 0.2 64
Validation accuracy:  0.466
0.001 0.5 0.2 200
Validation accuracy:  0.398
0.001 0.5 0.2 500
Validation accuracy:  0.301
0.001 0.5 0.25 64
Validation accuracy:  0.474
0.001 0.5 0.25 200
```

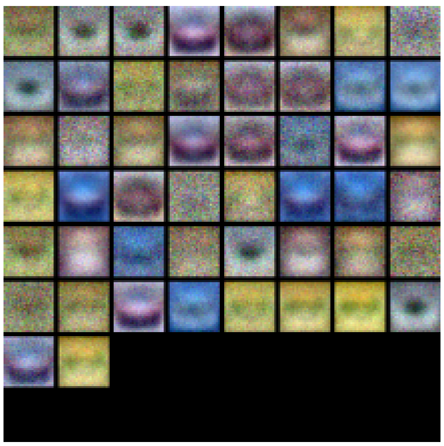
```
Validation accuracy:  0.413
0.001 0.5 0.25 500
Validation accuracy:  0.302
0.001 0.5 0.5 64
Validation accuracy:  0.45
0.001 0.5 0.5 200
Validation accuracy:  0.394
0.001 0.5 0.5 500
Validation accuracy:  0.306
0.001 0.5 0.9 64
Validation accuracy:  0.441
0.001 0.5 0.9 200
Validation accuracy:  0.397
0.001 0.5 0.9 500
Validation accuracy:  0.288
0.001 0.5 0.99 64
Validation accuracy:  0.478
0.001 0.5 0.99 200
Validation accuracy:  0.397
0.001 0.5 0.99 500
Validation accuracy:  0.285
0.0004641588833612777 0.99 0 64
Validation accuracy:  0.49
0.0004641588833612777 0.99 0 200
Validation accuracy:  0.484
0.0004641588833612777 0.99 0 500
Validation accuracy:  0.513
0.0004641588833612777 0.99 0.05 64
Validation accuracy:  0.488
0.0004641588833612777 0.99 0.05 200
Validation accuracy:  0.506
0.0004641588833612777 0.99 0.05 500
Validation accuracy:  0.492
0.0004641588833612777 0.99 0.1 64
Validation accuracy:  0.483
0.0004641588833612777 0.99 0.1 200
Validation accuracy:  0.502
0.0004641588833612777 0.99 0.1 500
Validation accuracy:  0.497
0.0004641588833612777 0.99 0.2 64
Validation accuracy:  0.491
0.0004641588833612777 0.99 0.2 200
Validation accuracy:  0.495
0.0004641588833612777 0.99 0.2 500
Validation accuracy:  0.502
0.0004641588833612777 0.99 0.25 64
Validation accuracy:  0.483
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-13-6ae0aa099f48> in <module>
     32                             num_iters=3000, batch_size=batch_size,
     33                             learning_rate=learning_rate, learning_rat
e_decay=learning_rate_decay,
---> 34                             reg=reg, verbose=False)
     35 
     36                 # Predict on the validation set

~\Desktop\UCLA\ECENGR247C-80\HW3-code\nndl\neural_net.py in train(self, X, y,
X_val, y_val, learning_rate, learning_rate_decay, reg, num_iters, batch_size,
verbose)
    191         # ==========================================================
======= #
    192       idx_batch = np.random.choice(num_train,batch_size)
--> 193       X_batch = X[idx_batch,:]
    194       y_batch = y[idx_batch]
    195       pass

KeyboardInterrupt: 
```

In [14]:
```
print('Best Validation accuracy: ', best_acc)
```

```
Best Validation accuracy:  0.522
```

In [15]:
```
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) In the suboptimal net, I see some vague templates of objects such as a car, some blue blobs for ocean/sky, and some bright green blobs for grass (although mostly random pixel coloration), but for the best net that I arrived at templates are much more clear (a car is clearly visible, much more blue for ocean/sky, green appears more for grass, and less random pixel coloration with more solid color backgrounds).

Evaluate on test set

In [16]:

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

```
Test accuracy:  0.501
```