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

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:
3.381231233889892e-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 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=False)
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, grads[p
```

```
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
```

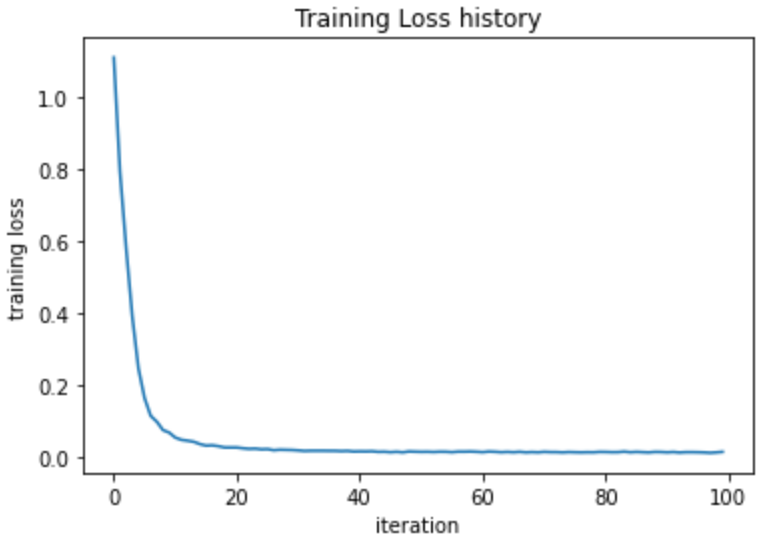
Training the network

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

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:  0.014498902952971647
```



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 = '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 [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 1.94651768178565
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

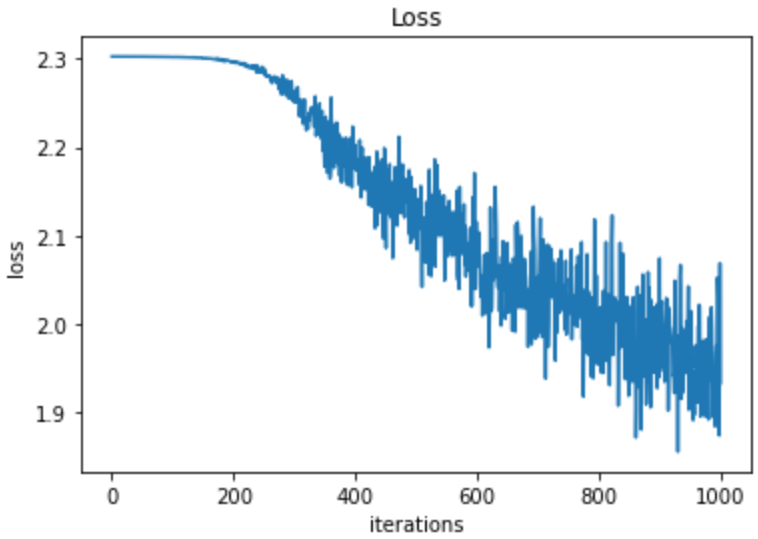
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

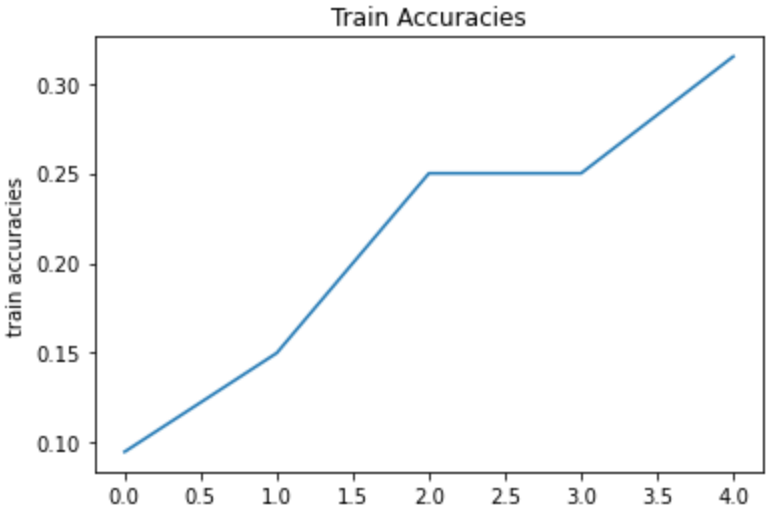
(2) How should you fix the problems you identified in (1)?

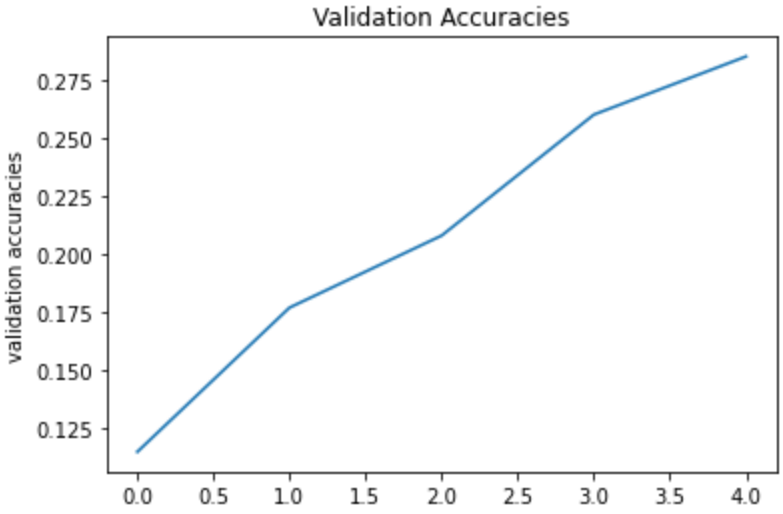
In [10]:
```
stats['train_acc_history']
```

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

plt.plot(stats['train_acc_history'])
plt.ylabel('train accuracies')
plt.title('Train Accuracies')
plt.show()

plt.plot(stats['val_acc_history'])
plt.ylabel('validation accuracies')
plt.title('Validation Accuracies')
plt.show()

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







Answers:

(1) From the loss plot, the loss didn't decrease at the beginning. Therefore, we can guess that the learning rate is too small. From the two accuracy plots, we can see that the accuracies are sill rising. Hence, we can guess that 1000 iterations may not be enough.

(2) I will increase the learning rate and the 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]:

```
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)!
# ================================================================ #

learning_rates = [1e-4, 1e-3, 3e-3, 5e-3, 1e-2, 3e-2, 5e-2, 1e-1]
accuracy = {}

best_learning_rate = 0
best_validation = 0

for learning_rate in learning_rates:
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=3500, batch_size=200,
            learning_rate=learning_rate, learning_rate_decay=0.95,
            reg=0.55, verbose=True)

    y_train_pred = net.predict(X_train)
    train_accuracy = np.mean(np.equal(y_train, y_train_pred))

    y_val_pred = net.predict(X_val)
    val_accuracy = np.mean(np.equal(y_val, y_val_pred))

    accuracy[learning_rate] = (train_accuracy, val_accuracy)

    if best_validation < val_accuracy:
        best_learning_rate = learning_rate
        best_validation = val_accuracy
        best_net = net

for learning_rate in accuracy:
    print("Learning Rate: {}, Train Accuracy: {}, Validation: {}".format(learning_rate, ac

print("\nThe Best Learning Rate: {}\n".format(best_learning_rate))

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 3500: loss 2.302997401109704
iteration 100 / 3500: loss 2.3025337778391846
iteration 200 / 3500: loss 2.298359174811698
iteration 300 / 3500: loss 2.264620168811341
iteration 400 / 3500: loss 2.2317722560179187
iteration 500 / 3500: loss 2.1473953573120252
iteration 600 / 3500: loss 2.078121763752046
iteration 700 / 3500: loss 2.045704612501719
iteration 800 / 3500: loss 1.965984706144976
```

```
iteration 900 / 3500: loss 2.014275621419568
iteration 1000 / 3500: loss 1.8921575161673971
iteration 1100 / 3500: loss 1.8598477867763987
iteration 1200 / 3500: loss 1.9298727425611475
iteration 1300 / 3500: loss 1.8962509476741416
iteration 1400 / 3500: loss 1.8794426275753993
iteration 1500 / 3500: loss 1.8120413442682877
iteration 1600 / 3500: loss 1.8291193208791063
iteration 1700 / 3500: loss 1.9029972517200149
iteration 1800 / 3500: loss 1.6872365256217474
iteration 1900 / 3500: loss 1.911582176437726
iteration 2000 / 3500: loss 1.7462618983914069
iteration 2100 / 3500: loss 1.757088388209533
iteration 2200 / 3500: loss 1.7543581198803018
iteration 2300 / 3500: loss 1.7144871769537742
iteration 2400 / 3500: loss 1.6471536306229593
iteration 2500 / 3500: loss 1.7821161425180871
iteration 2600 / 3500: loss 1.8573546497173177
iteration 2700 / 3500: loss 1.8141297778440546
iteration 2800 / 3500: loss 1.7338615965988042
iteration 2900 / 3500: loss 1.7435264603180365
iteration 3000 / 3500: loss 1.699752449476504
iteration 3100 / 3500: loss 1.8324479342243916
iteration 3200 / 3500: loss 1.7612130378984054
iteration 3300 / 3500: loss 1.7218378406578894
iteration 3400 / 3500: loss 1.6993344012142404
iteration 0 / 3500: loss 2.302986411324343
iteration 100 / 3500: loss 1.9232232372666749
iteration 200 / 3500: loss 1.759347222135508
iteration 300 / 3500: loss 1.756488564219933
iteration 400 / 3500: loss 1.6727663289497905
iteration 500 / 3500: loss 1.633897580429175
iteration 600 / 3500: loss 1.678366978962524
iteration 700 / 3500: loss 1.5908599823897758
iteration 800 / 3500: loss 1.4762522293413398
iteration 900 / 3500: loss 1.53800150752706
iteration 1000 / 3500: loss 1.576294915354204
iteration 1100 / 3500: loss 1.488414553954279
iteration 1200 / 3500: loss 1.3870829853306392
iteration 1300 / 3500: loss 1.5789796596586665
iteration 1400 / 3500: loss 1.4333352374069301
iteration 1500 / 3500: loss 1.3348892324102062
iteration 1600 / 3500: loss 1.4882764475636145
iteration 1700 / 3500: loss 1.5386421917250106
iteration 1800 / 3500: loss 1.4865942997101596
iteration 1900 / 3500: loss 1.5243896200801552
iteration 2000 / 3500: loss 1.4723743667183031
iteration 2100 / 3500: loss 1.5392486542155455
iteration 2200 / 3500: loss 1.446332157494875
iteration 2300 / 3500: loss 1.4651521029458334
iteration 2400 / 3500: loss 1.4213970746406865
iteration 2500 / 3500: loss 1.5286906235240358
iteration 2600 / 3500: loss 1.4881782566544015
iteration 2700 / 3500: loss 1.3928417400520017
iteration 2800 / 3500: loss 1.3581560073462011
iteration 2900 / 3500: loss 1.3760299236515272
iteration 3000 / 3500: loss 1.3836823654682613
iteration 3100 / 3500: loss 1.3123833713906103
iteration 3200 / 3500: loss 1.462576568868863
iteration 3300 / 3500: loss 1.3868367750092365
iteration 3400 / 3500: loss 1.5591674124610215
iteration 0 / 3500: loss 2.303014357893828
iteration 100 / 3500: loss 1.7068276303772936
iteration 200 / 3500: loss 1.7414195800082797
iteration 300 / 3500: loss 1.7519837577366806
iteration 400 / 3500: loss 1.6578147651681054
```

```
iteration 500 / 3500: loss 1.7700501946492917
iteration 600 / 3500: loss 1.6070792781069048
iteration 700 / 3500: loss 1.7733688800892693
iteration 800 / 3500: loss 1.7319000906465882
iteration 900 / 3500: loss 1.5989403728534124
iteration 1000 / 3500: loss 1.5344029574444085
iteration 1100 / 3500: loss 1.9133985971316425
iteration 1200 / 3500: loss 1.68329616121571
iteration 1300 / 3500: loss 1.464549162163786
iteration 1400 / 3500: loss 1.5971767049298458
iteration 1500 / 3500: loss 1.5251982076559232
iteration 1600 / 3500: loss 1.5783437880344389
iteration 1700 / 3500: loss 1.5252112516453449
iteration 1800 / 3500: loss 1.5204179087252612
iteration 1900 / 3500: loss 1.6369273560697226
iteration 2000 / 3500: loss 1.7068115558302446
iteration 2100 / 3500: loss 1.6126205076589328
iteration 2200 / 3500: loss 1.582731622426254
iteration 2300 / 3500: loss 1.5754925872045902
iteration 2400 / 3500: loss 1.701496290072023
iteration 2500 / 3500: loss 1.526950587945974
iteration 2600 / 3500: loss 1.5014990142459348
iteration 2700 / 3500: loss 1.5347069926852663
iteration 2800 / 3500: loss 1.6185106216349627
iteration 2900 / 3500: loss 1.5708537119911778
iteration 3000 / 3500: loss 1.5657797880625761
iteration 3100 / 3500: loss 1.557372436470082
iteration 3200 / 3500: loss 1.307196726686845
iteration 3300 / 3500: loss 1.3896861393043467
iteration 3400 / 3500: loss 1.4343703894886384
iteration 0 / 3500: loss 2.303012541137858
/Users/jacky/My_Data/Data/UCLA/2022_Winter/ECE C247_Deep Learning/Homework/HW3/hw3-code/nn
dl/neural_net.py:115: RuntimeWarning: divide by zero encountered in log
  probs_log = -np.log(probs_row)
iteration 100 / 3500: loss inf
iteration 200 / 3500: loss inf
iteration 300 / 3500: loss inf
iteration 400 / 3500: loss inf
iteration 500 / 3500: loss inf
iteration 600 / 3500: loss inf
iteration 700 / 3500: loss inf
iteration 800 / 3500: loss inf
iteration 900 / 3500: loss inf
/Users/jacky/My_Data/Data/UCLA/2022_Winter/ECE C247_Deep Learning/Homework/HW3/hw3-code/nn
dl/neural_net.py:110: RuntimeWarning: overflow encountered in subtract
  scores -= np.max(scores, axis=1, keepdims=True)
/Users/jacky/My_Data/Data/UCLA/2022_Winter/ECE C247_Deep Learning/Homework/HW3/hw3-code/nn
dl/neural_net.py:110: RuntimeWarning: invalid value encountered in subtract
  scores -= np.max(scores, axis=1, keepdims=True)
iteration 1000 / 3500: loss nan
iteration 1100 / 3500: loss nan
iteration 1200 / 3500: loss nan
iteration 1300 / 3500: loss nan
iteration 1400 / 3500: loss nan
iteration 1500 / 3500: loss nan
iteration 1600 / 3500: loss nan
iteration 1700 / 3500: loss nan
iteration 1800 / 3500: loss nan
iteration 1900 / 3500: loss nan
iteration 2000 / 3500: loss nan
iteration 2100 / 3500: loss nan
iteration 2200 / 3500: loss nan
iteration 2300 / 3500: loss nan
iteration 2400 / 3500: loss nan
iteration 2500 / 3500: loss nan
```

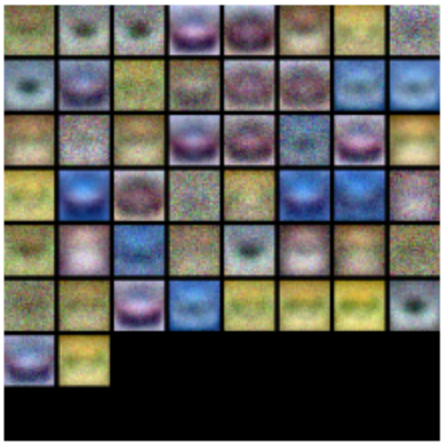
```
iteration 2600 / 3500: loss nan
iteration 2700 / 3500: loss nan
iteration 2800 / 3500: loss nan
iteration 2900 / 3500: loss nan
iteration 3000 / 3500: loss nan
iteration 3100 / 3500: loss nan
iteration 3200 / 3500: loss nan
iteration 3300 / 3500: loss nan
iteration 3400 / 3500: loss nan
iteration 0 / 3500: loss 2.303024951829114
iteration 100 / 3500: loss inf
iteration 200 / 3500: loss inf
iteration 300 / 3500: loss inf
iteration 400 / 3500: loss nan
iteration 500 / 3500: loss nan
iteration 600 / 3500: loss nan
iteration 700 / 3500: loss nan
iteration 800 / 3500: loss nan
iteration 900 / 3500: loss nan
iteration 1000 / 3500: loss nan
iteration 1100 / 3500: loss nan
iteration 1200 / 3500: loss nan
iteration 1300 / 3500: loss nan
iteration 1400 / 3500: loss nan
iteration 1500 / 3500: loss nan
iteration 1600 / 3500: loss nan
iteration 1700 / 3500: loss nan
iteration 1800 / 3500: loss nan
iteration 1900 / 3500: loss nan
iteration 2000 / 3500: loss nan
iteration 2100 / 3500: loss nan
iteration 2200 / 3500: loss nan
iteration 2300 / 3500: loss nan
iteration 2400 / 3500: loss nan
iteration 2500 / 3500: loss nan
iteration 2600 / 3500: loss nan
iteration 2700 / 3500: loss nan
iteration 2800 / 3500: loss nan
iteration 2900 / 3500: loss nan
iteration 3000 / 3500: loss nan
iteration 3100 / 3500: loss nan
iteration 3200 / 3500: loss nan
iteration 3300 / 3500: loss nan
iteration 3400 / 3500: loss nan
iteration 0 / 3500: loss 2.303007161998417
iteration 100 / 3500: loss inf
iteration 200 / 3500: loss nan
iteration 300 / 3500: loss nan
iteration 400 / 3500: loss nan
iteration 500 / 3500: loss nan
iteration 600 / 3500: loss nan
iteration 700 / 3500: loss nan
iteration 800 / 3500: loss nan
iteration 900 / 3500: loss nan
iteration 1000 / 3500: loss nan
iteration 1100 / 3500: loss nan
iteration 1200 / 3500: loss nan
iteration 1300 / 3500: loss nan
iteration 1400 / 3500: loss nan
iteration 1500 / 3500: loss nan
iteration 1600 / 3500: loss nan
iteration 1700 / 3500: loss nan
iteration 1800 / 3500: loss nan
iteration 1900 / 3500: loss nan
iteration 2000 / 3500: loss nan
iteration 2100 / 3500: loss nan
```

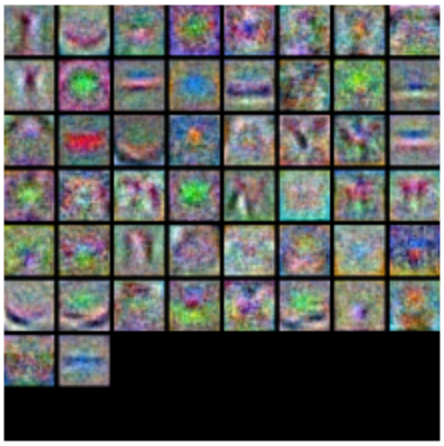
```
iteration 2200 / 3500: loss nan
iteration 2300 / 3500: loss nan
iteration 2400 / 3500: loss nan
iteration 2500 / 3500: loss nan
iteration 2600 / 3500: loss nan
iteration 2700 / 3500: loss nan
iteration 2800 / 3500: loss nan
iteration 2900 / 3500: loss nan
iteration 3000 / 3500: loss nan
iteration 3100 / 3500: loss nan
iteration 3200 / 3500: loss nan
iteration 3300 / 3500: loss nan
iteration 3400 / 3500: loss nan
iteration 0 / 3500: loss 2.3030106917108073
iteration 100 / 3500: loss inf
iteration 200 / 3500: loss nan
iteration 300 / 3500: loss nan
iteration 400 / 3500: loss nan
iteration 500 / 3500: loss nan
iteration 600 / 3500: loss nan
iteration 700 / 3500: loss nan
iteration 800 / 3500: loss nan
iteration 900 / 3500: loss nan
iteration 1000 / 3500: loss nan
iteration 1100 / 3500: loss nan
iteration 1200 / 3500: loss nan
iteration 1300 / 3500: loss nan
iteration 1400 / 3500: loss nan
iteration 1500 / 3500: loss nan
iteration 1600 / 3500: loss nan
iteration 1700 / 3500: loss nan
iteration 1800 / 3500: loss nan
iteration 1900 / 3500: loss nan
iteration 2000 / 3500: loss nan
iteration 2100 / 3500: loss nan
iteration 2200 / 3500: loss nan
iteration 2300 / 3500: loss nan
iteration 2400 / 3500: loss nan
iteration 2500 / 3500: loss nan
iteration 2600 / 3500: loss nan
iteration 2700 / 3500: loss nan
iteration 2800 / 3500: loss nan
iteration 2900 / 3500: loss nan
iteration 3000 / 3500: loss nan
iteration 3100 / 3500: loss nan
iteration 3200 / 3500: loss nan
iteration 3300 / 3500: loss nan
iteration 3400 / 3500: loss nan
iteration 0 / 3500: loss 2.3030054797882737
iteration 100 / 3500: loss inf
iteration 200 / 3500: loss nan
iteration 300 / 3500: loss nan
iteration 400 / 3500: loss nan
iteration 500 / 3500: loss nan
iteration 600 / 3500: loss nan
iteration 700 / 3500: loss nan
iteration 800 / 3500: loss nan
iteration 900 / 3500: loss nan
iteration 1000 / 3500: loss nan
iteration 1100 / 3500: loss nan
iteration 1200 / 3500: loss nan
iteration 1300 / 3500: loss nan
iteration 1400 / 3500: loss nan
iteration 1500 / 3500: loss nan
iteration 1600 / 3500: loss nan
iteration 1700 / 3500: loss nan
```

```
iteration 1800 / 3500: loss nan
iteration 1900 / 3500: loss nan
iteration 2000 / 3500: loss nan
iteration 2100 / 3500: loss nan
iteration 2200 / 3500: loss nan
iteration 2300 / 3500: loss nan
iteration 2400 / 3500: loss nan
iteration 2500 / 3500: loss nan
iteration 2600 / 3500: loss nan
iteration 2700 / 3500: loss nan
iteration 2800 / 3500: loss nan
iteration 2900 / 3500: loss nan
iteration 3000 / 3500: loss nan
iteration 3100 / 3500: loss nan
iteration 3200 / 3500: loss nan
iteration 3300 / 3500: loss nan
iteration 3400 / 3500: loss nan
Learning Rate: 0.0001, Train Accuracy: 0.397734693877551, Validation: 0.389
Learning Rate: 0.001, Train Accuracy: 0.5489387755102041, Validation: 0.496
Learning Rate: 0.003, Train Accuracy: 0.5375918367346939, Validation: 0.502
Learning Rate: 0.005, Train Accuracy: 0.10026530612244898, Validation: 0.087
Learning Rate: 0.01, Train Accuracy: 0.10026530612244898, Validation: 0.087
Learning Rate: 0.03, Train Accuracy: 0.10026530612244898, Validation: 0.087
Learning Rate: 0.05, Train Accuracy: 0.10026530612244898, Validation: 0.087
Learning Rate: 0.1, Train Accuracy: 0.10026530612244898, Validation: 0.087

The Best Learning Rate: 0.003

Validation accuracy:  0.502
```

In [13]:

```
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 images in the suboptimal net looks alike, and they contain much more noises. However, we can distinguish the differences between the images in the best net.

Evaluate on test set

In [14]:

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

```
Test accuracy:  0.513
```