two_layer_nn

February 3, 2021

0.1 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.

[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))))
```

0.2 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass

[2]:
```
from nndl.neural_net import TwoLayerNet
```

[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()
```

0.2.1 Compute forward pass scores

[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
```

0.2.2 Forward pass loss

[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
```

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

```
1.071696123862817
```

0.2.3 Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

[7]:
```
from cs231n.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[param_name])))
```

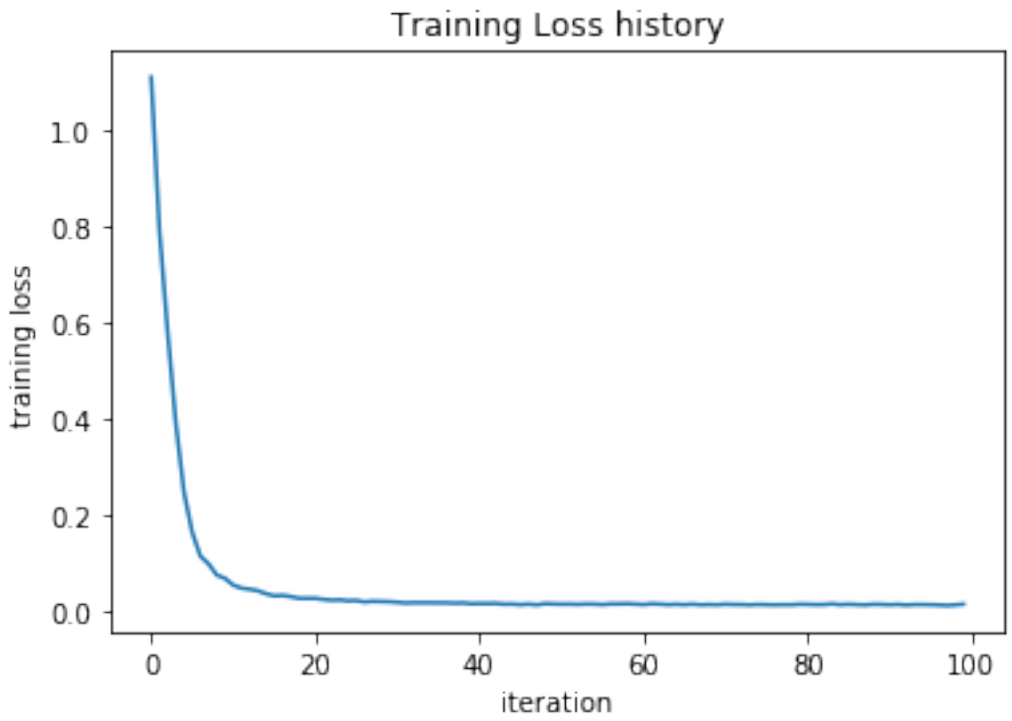
```
W2 max relative error: 2.9632245016399034e-10
b2 max relative error: 1.8392106647421603e-10
W1 max relative error: 1.2832892417669998e-09
b1 max relative error: 3.172680285697327e-09
```

0.2.4 Training the network

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

[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:  0.014497865475252903
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[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 = '/Users/stathismegas/Documents/ECE_247/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,)
```

0.3.1 Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

[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 2.302120160170799
iteration 200 / 1000: loss 2.2956136025577862
iteration 300 / 1000: loss 2.2518259124925293
iteration 400 / 1000: loss 2.1889952557361934
iteration 500 / 1000: loss 2.1162528247006445
iteration 600 / 1000: loss 2.0646709106774503
iteration 700 / 1000: loss 1.9901688837884741
iteration 800 / 1000: loss 2.0028277559728718
iteration 900 / 1000: loss 1.9465178635514782
Validation accuracy:  0.283
```

0.4 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.

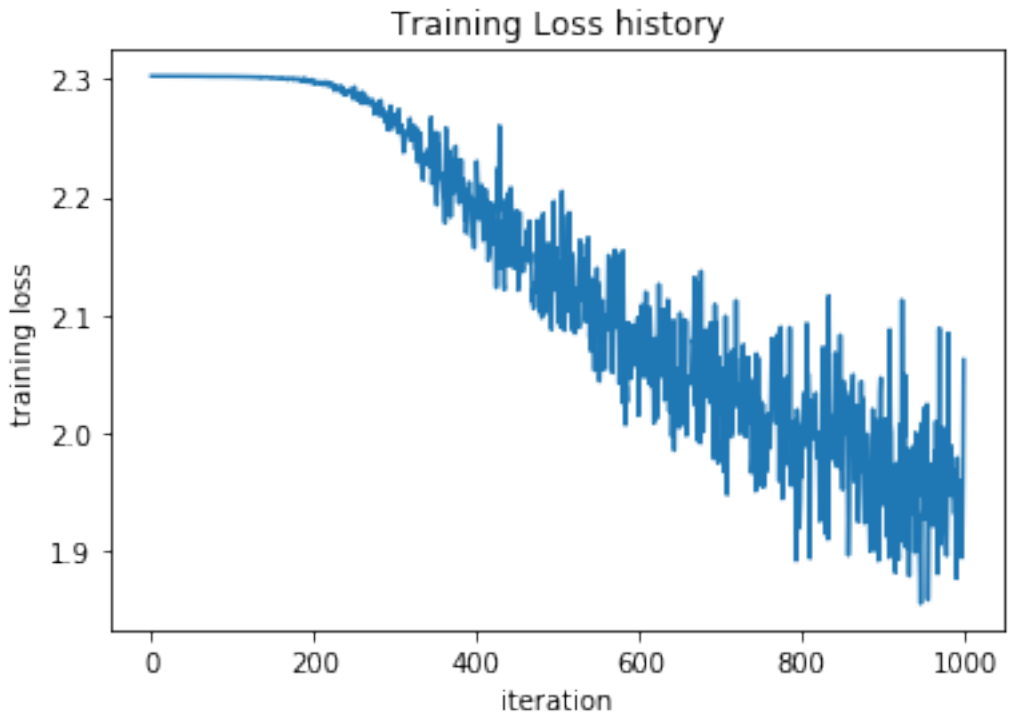
(2) How should you fix the problems you identified in (1)?

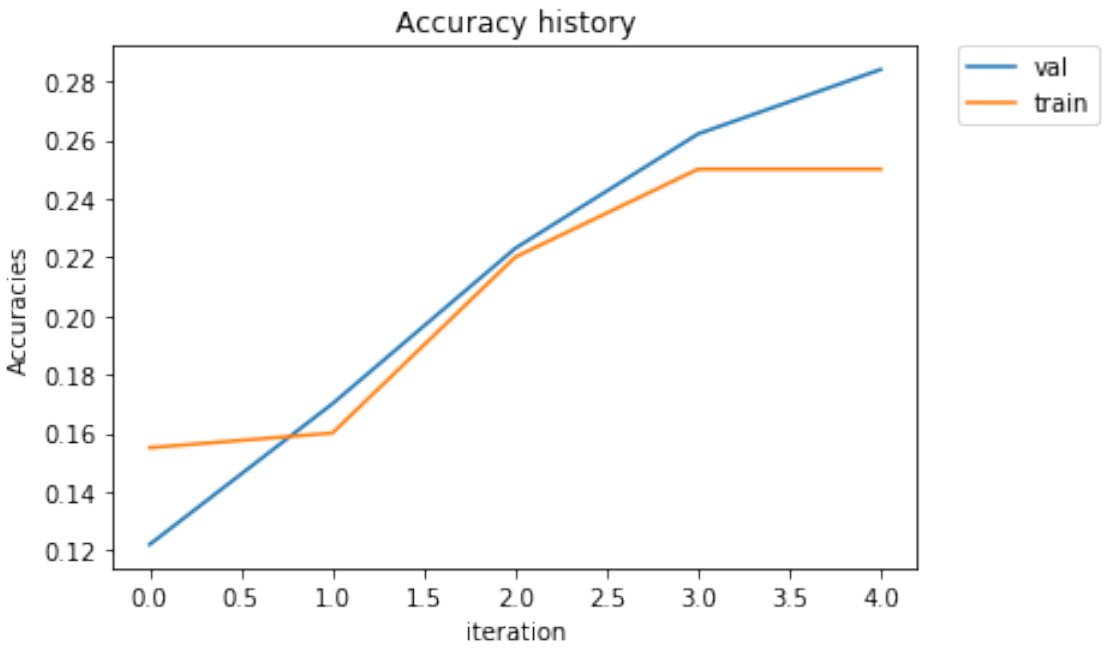
[11]:
```
stats['train_acc_history']
```

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

[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

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
plt.plot( stats['val_acc_history'], label="val")
plt.plot(stats['train_acc_history'], label="train")
plt.xlabel('iteration')
plt.ylabel('Accuracies')
plt.title('Accuracy history')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()

# f = plt.figure()
# ax = f.gca()
# ax.plot(stats['train_acc_history'], stats['val_acc_history'], '-o')
# ax.set_xlabel('$iteration$')
# ax.set_ylabel('$accuracy$')

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





0.5 Answers:

(1) We see that the loss function is very noisy with a mean which is roughly a straight line after iteration 300. These two features are a hint that the learning rate is small! We would want the initial exponential decay of the loss to persist longer so we need to increase the learning rate. Likely it also means we should adjust the number of iterations and the learning_rate_decay. Moreover, we see that the accuracies keep increasing and they haven't saturated. Yet both accuracies are so low that it is clear that the learning rate, number of iterations and learning rate decay need adjustment.

(2) The way to perform hyperparameter tuning is to train the model of the train data for different specifications of the hyperparameters, then calculate the validation accuracy, and finally pick the hyperparamters that had the maximum validation accuracy. We do this below.

0.6 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.

[30]:
```
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)

rates = [ 9e-4, 1e-3, 1.1e-3, 1.2e-3, 1.4e-3, 1.5e-3, 1.6e-3, 1.8e-3, 2e-3␣
↪,1e-4]
decays = [0.91, 0.92, 0.93]
best=0
best_rate=0
best_decay=0

for decay in decays:
    for rate in rates:
        # Train the network
        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=rate, learning_rate_decay=decay,
            reg=0.25, verbose=True)

        # Predict on the validation set
        val_acc = (net.predict(X_val) == y_val).mean()
        print('Validation accuracy: ', val_acc, 'for learning rate=', rate,'and␣
↪decay rate=', decay)

        if val_acc>best:
            best = val_acc
            best_net = net
            best_rate = rate
            best_decay = decay


print('best validation accuracy ', best, 'achieved for learning rate',␣
↪best_rate, 'and decay', best_decay)


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

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

```
iteration 0 / 1000: loss 2.3027526263458826
iteration 100 / 1000: loss 2.037422584023342
iteration 200 / 1000: loss 1.8063172829654737
iteration 300 / 1000: loss 1.7266531403359007
iteration 400 / 1000: loss 1.7117307632715548
iteration 500 / 1000: loss 1.6523430829766477
iteration 600 / 1000: loss 1.6176053001213357
iteration 700 / 1000: loss 1.5520675504820995
iteration 800 / 1000: loss 1.473717380809065
iteration 900 / 1000: loss 1.4987916655599491
Validation accuracy:  0.455 for learning rate= 0.0009 and decay rate= 0.91
iteration 0 / 1000: loss 2.302764747712296
iteration 100 / 1000: loss 2.0551035766260437
iteration 200 / 1000: loss 1.805449265167068
iteration 300 / 1000: loss 1.7158471167141658
iteration 400 / 1000: loss 1.668332818026018
iteration 500 / 1000: loss 1.5329187156511352
iteration 600 / 1000: loss 1.5529101232342561
iteration 700 / 1000: loss 1.4913610314197687
iteration 800 / 1000: loss 1.5817636881214792
iteration 900 / 1000: loss 1.5279816895403542
Validation accuracy:  0.461 for learning rate= 0.001 and decay rate= 0.91
iteration 0 / 1000: loss 2.302766852272655
iteration 100 / 1000: loss 1.855235840033207
iteration 200 / 1000: loss 1.8675844283360492
iteration 300 / 1000: loss 1.7178357306705292
iteration 400 / 1000: loss 1.6437015113766227
iteration 500 / 1000: loss 1.4671104184796173
iteration 600 / 1000: loss 1.6267537139398216
iteration 700 / 1000: loss 1.5753304871589
iteration 800 / 1000: loss 1.5612237028478564
iteration 900 / 1000: loss 1.5764734165392968
Validation accuracy:  0.465 for learning rate= 0.0011 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027764598997176
iteration 100 / 1000: loss 1.9626617441087344
iteration 200 / 1000: loss 1.7642765066552624
iteration 300 / 1000: loss 1.690622408728896
iteration 400 / 1000: loss 1.6502610745165731
iteration 500 / 1000: loss 1.4860774448938399
iteration 600 / 1000: loss 1.4888684224217603
iteration 700 / 1000: loss 1.5057732642153925
iteration 800 / 1000: loss 1.6079413630666022
iteration 900 / 1000: loss 1.5482142684063374
Validation accuracy:  0.483 for learning rate= 0.0012 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027811931269064
```

```
iteration 100 / 1000: loss 1.9082413493381
iteration 200 / 1000: loss 1.6902834536098321
iteration 300 / 1000: loss 1.4995754285038871
iteration 400 / 1000: loss 1.6765854657150168
iteration 500 / 1000: loss 1.5754334182647751
iteration 600 / 1000: loss 1.6382714697352592
iteration 700 / 1000: loss 1.5389399558512489
iteration 800 / 1000: loss 1.4386463455227754
iteration 900 / 1000: loss 1.3965520887781604
Validation accuracy:  0.476 for learning rate= 0.0014 and decay rate= 0.91
iteration 0 / 1000: loss 2.302779184717079
iteration 100 / 1000: loss 1.8741982028033704
iteration 200 / 1000: loss 1.7245724788786345
iteration 300 / 1000: loss 1.5620967493973503
iteration 400 / 1000: loss 1.6136589336123084
iteration 500 / 1000: loss 1.6406302083322952
iteration 600 / 1000: loss 1.507580058353166
iteration 700 / 1000: loss 1.604260452416418
iteration 800 / 1000: loss 1.5319333393445522
iteration 900 / 1000: loss 1.564120763222298
Validation accuracy:  0.464 for learning rate= 0.0015 and decay rate= 0.91
iteration 0 / 1000: loss 2.30280608437208
iteration 100 / 1000: loss 1.844347668160187
iteration 200 / 1000: loss 1.6693091159577698
iteration 300 / 1000: loss 1.6417388097343981
iteration 400 / 1000: loss 1.6450387854298085
iteration 500 / 1000: loss 1.4311272798917978
iteration 600 / 1000: loss 1.446066499037622
iteration 700 / 1000: loss 1.4438185708296873
iteration 800 / 1000: loss 1.473262465633535
iteration 900 / 1000: loss 1.3780585344469705
Validation accuracy:  0.489 for learning rate= 0.0016 and decay rate= 0.91
iteration 0 / 1000: loss 2.302750861249898
iteration 100 / 1000: loss 1.819096005456171
iteration 200 / 1000: loss 1.6986503091863674
iteration 300 / 1000: loss 1.6575175319812596
iteration 400 / 1000: loss 1.7330169264919373
iteration 500 / 1000: loss 1.407538264143662
iteration 600 / 1000: loss 1.449062675963313
iteration 700 / 1000: loss 1.4881309966281373
iteration 800 / 1000: loss 1.4852235479224032
iteration 900 / 1000: loss 1.5832232839697862
Validation accuracy:  0.472 for learning rate= 0.0018 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027764803930637
iteration 100 / 1000: loss 1.7666360596126112
iteration 200 / 1000: loss 1.6661962752947854
iteration 300 / 1000: loss 1.6999386351179402
iteration 400 / 1000: loss 1.5768473862582923
```

```
iteration 500 / 1000: loss 1.5625661897692444
iteration 600 / 1000: loss 1.731500669467699
iteration 700 / 1000: loss 1.5758523547277488
iteration 800 / 1000: loss 1.5080668710421075
iteration 900 / 1000: loss 1.3670808491281907
Validation accuracy:  0.475 for learning rate= 0.002 and decay rate= 0.91
iteration 0 / 1000: loss 2.3027886031861735
iteration 100 / 1000: loss 2.302533871368175
iteration 200 / 1000: loss 2.2995890510324717
iteration 300 / 1000: loss 2.274271170412578
iteration 400 / 1000: loss 2.25020365022876
iteration 500 / 1000: loss 2.2002144505112735
iteration 600 / 1000: loss 2.134826036920963
iteration 700 / 1000: loss 2.089515609506605
iteration 800 / 1000: loss 1.9707954594611419
iteration 900 / 1000: loss 2.030086299887209
Validation accuracy:  0.274 for learning rate= 0.0001 and decay rate= 0.91
iteration 0 / 1000: loss 2.302765225758387
iteration 100 / 1000: loss 2.0456681259091605
iteration 200 / 1000: loss 1.786443653581439
iteration 300 / 1000: loss 1.6945349592896
iteration 400 / 1000: loss 1.6538275805435
iteration 500 / 1000: loss 1.632368840740618
iteration 600 / 1000: loss 1.5050096224608955
iteration 700 / 1000: loss 1.5423219198786622
iteration 800 / 1000: loss 1.556037673841548
iteration 900 / 1000: loss 1.4199593816582998
Validation accuracy:  0.457 for learning rate= 0.0009 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027875440096817
iteration 100 / 1000: loss 1.893192465425405
iteration 200 / 1000: loss 1.7166711303251012
iteration 300 / 1000: loss 1.7479828400383337
iteration 400 / 1000: loss 1.5818909354310782
iteration 500 / 1000: loss 1.533252403371238
iteration 600 / 1000: loss 1.6675537965822993
iteration 700 / 1000: loss 1.4628574994586232
iteration 800 / 1000: loss 1.5234503808913165
iteration 900 / 1000: loss 1.4923782729519934
Validation accuracy:  0.488 for learning rate= 0.001 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027879282421972
iteration 100 / 1000: loss 1.8347423406391759
iteration 200 / 1000: loss 1.8311958227659326
iteration 300 / 1000: loss 1.572280395078183
iteration 400 / 1000: loss 1.7540775583742172
iteration 500 / 1000: loss 1.475113698667047
iteration 600 / 1000: loss 1.7005921772815111
iteration 700 / 1000: loss 1.568302438100576
iteration 800 / 1000: loss 1.4255591861318369
```

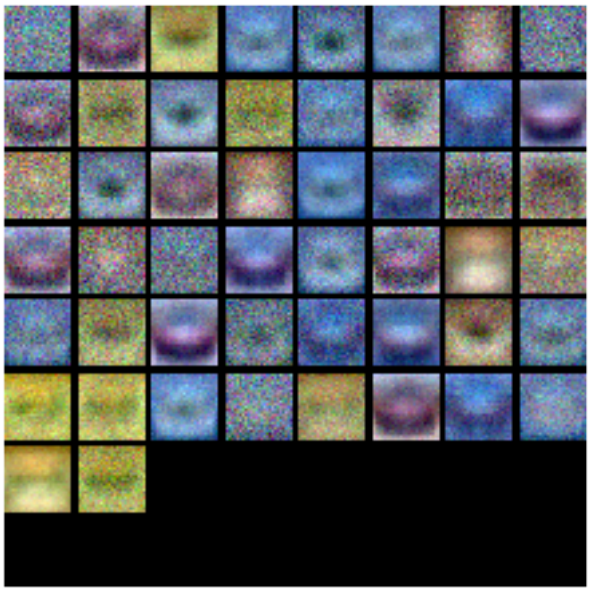
```
iteration 900 / 1000: loss 1.4214717842026463
Validation accuracy:  0.481 for learning rate= 0.0011 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027877966236256
iteration 100 / 1000: loss 1.9025250585142668
iteration 200 / 1000: loss 1.7845374628555084
iteration 300 / 1000: loss 1.7891133134763677
iteration 400 / 1000: loss 1.6250452612311737
iteration 500 / 1000: loss 1.5456332798448067
iteration 600 / 1000: loss 1.5794498909274866
iteration 700 / 1000: loss 1.5152706977299364
iteration 800 / 1000: loss 1.525523299029692
iteration 900 / 1000: loss 1.3821067009338521
Validation accuracy:  0.48 for learning rate= 0.0012 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027821195137346
iteration 100 / 1000: loss 1.815597285691698
iteration 200 / 1000: loss 1.7991268879709077
iteration 300 / 1000: loss 1.666103974818494
iteration 400 / 1000: loss 1.6905994971634697
iteration 500 / 1000: loss 1.4756411125111968
iteration 600 / 1000: loss 1.5687947634261603
iteration 700 / 1000: loss 1.4782461627557029
iteration 800 / 1000: loss 1.547258877718856
iteration 900 / 1000: loss 1.5621352272609677
Validation accuracy:  0.472 for learning rate= 0.0014 and decay rate= 0.92
iteration 0 / 1000: loss 2.302782568036341
iteration 100 / 1000: loss 1.835301814385985
iteration 200 / 1000: loss 1.6799214174151353
iteration 300 / 1000: loss 1.6160132102638458
iteration 400 / 1000: loss 1.7358092682569097
iteration 500 / 1000: loss 1.552820431337074
iteration 600 / 1000: loss 1.6005060273751806
iteration 700 / 1000: loss 1.5376882804793943
iteration 800 / 1000: loss 1.5762845037532875
iteration 900 / 1000: loss 1.4910685469870995
Validation accuracy:  0.461 for learning rate= 0.0015 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027620957625845
iteration 100 / 1000: loss 1.8185433142811709
iteration 200 / 1000: loss 1.6663917008525475
iteration 300 / 1000: loss 1.5054221375292076
iteration 400 / 1000: loss 1.5822511273494242
iteration 500 / 1000: loss 1.5361872019104705
iteration 600 / 1000: loss 1.5984936787144204
iteration 700 / 1000: loss 1.5001788008846235
iteration 800 / 1000: loss 1.5750661777506691
iteration 900 / 1000: loss 1.5304911021803427
Validation accuracy:  0.488 for learning rate= 0.0016 and decay rate= 0.92
iteration 0 / 1000: loss 2.30279132805707
iteration 100 / 1000: loss 1.8022569025215653
```

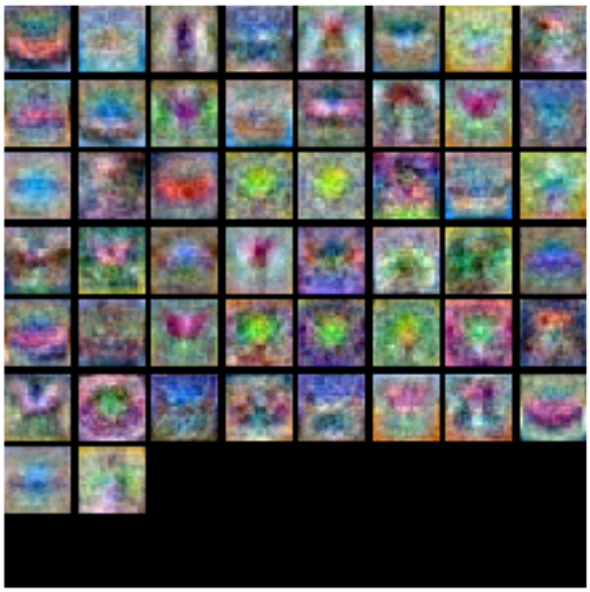
```
iteration 200 / 1000: loss 1.7573959921805067
iteration 300 / 1000: loss 1.6555398440790572
iteration 400 / 1000: loss 1.685691605374468
iteration 500 / 1000: loss 1.551463633368692
iteration 600 / 1000: loss 1.471125429922446
iteration 700 / 1000: loss 1.5437237629860001
iteration 800 / 1000: loss 1.4709821741433275
iteration 900 / 1000: loss 1.6465870701008538
Validation accuracy:  0.46 for learning rate= 0.0018 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027653782483903
iteration 100 / 1000: loss 1.7354278700991306
iteration 200 / 1000: loss 1.692773437371705
iteration 300 / 1000: loss 1.6308499124637141
iteration 400 / 1000: loss 1.6799614056389043
iteration 500 / 1000: loss 1.4751466940373366
iteration 600 / 1000: loss 1.7528690158745446
iteration 700 / 1000: loss 1.571636255433799
iteration 800 / 1000: loss 1.4913193076122098
iteration 900 / 1000: loss 1.505855628654246
Validation accuracy:  0.449 for learning rate= 0.002 and decay rate= 0.92
iteration 0 / 1000: loss 2.3027608591890325
iteration 100 / 1000: loss 2.302305779563855
iteration 200 / 1000: loss 2.2954228136909407
iteration 300 / 1000: loss 2.248648408026622
iteration 400 / 1000: loss 2.223361274912419
iteration 500 / 1000: loss 2.169656954955473
iteration 600 / 1000: loss 2.0937973509531065
iteration 700 / 1000: loss 2.0661813765817767
iteration 800 / 1000: loss 2.0725727880403833
iteration 900 / 1000: loss 1.9881720834217709
Validation accuracy:  0.273 for learning rate= 0.0001 and decay rate= 0.92
iteration 0 / 1000: loss 2.302799524865559
iteration 100 / 1000: loss 1.9940750813606323
iteration 200 / 1000: loss 1.7211028884783817
iteration 300 / 1000: loss 1.699737404568344
iteration 400 / 1000: loss 1.7850021763742054
iteration 500 / 1000: loss 1.6608943321713947
iteration 600 / 1000: loss 1.7556723024018612
iteration 700 / 1000: loss 1.6797959089158714
iteration 800 / 1000: loss 1.6125549391236447
iteration 900 / 1000: loss 1.4418816636118525
Validation accuracy:  0.471 for learning rate= 0.0009 and decay rate= 0.93
iteration 0 / 1000: loss 2.3028089084282795
iteration 100 / 1000: loss 1.9578561389940095
iteration 200 / 1000: loss 1.7532891891711948
iteration 300 / 1000: loss 1.6283618901230952
iteration 400 / 1000: loss 1.6670149341486005
iteration 500 / 1000: loss 1.531887080385329
```

```
iteration 600 / 1000: loss 1.5401651930568854
iteration 700 / 1000: loss 1.447107094577251
iteration 800 / 1000: loss 1.6555877622214972
iteration 900 / 1000: loss 1.6356135852796474
Validation accuracy:  0.471 for learning rate= 0.001 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027914774734963
iteration 100 / 1000: loss 1.9134514974450185
iteration 200 / 1000: loss 1.8190576140026622
iteration 300 / 1000: loss 1.6278760988062977
iteration 400 / 1000: loss 1.4848310488246041
iteration 500 / 1000: loss 1.6039628321171158
iteration 600 / 1000: loss 1.3854227668578198
iteration 700 / 1000: loss 1.3951224054104614
iteration 800 / 1000: loss 1.6405236444994489
iteration 900 / 1000: loss 1.3930023110969063
Validation accuracy:  0.478 for learning rate= 0.0011 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027834683215183
iteration 100 / 1000: loss 1.9976130905594969
iteration 200 / 1000: loss 1.6972729286003418
iteration 300 / 1000: loss 1.7222532972312177
iteration 400 / 1000: loss 1.5929367696983607
iteration 500 / 1000: loss 1.5402579996768127
iteration 600 / 1000: loss 1.5275985210170882
iteration 700 / 1000: loss 1.5595503840206457
iteration 800 / 1000: loss 1.5384125115052179
iteration 900 / 1000: loss 1.4632347069188691
Validation accuracy:  0.479 for learning rate= 0.0012 and decay rate= 0.93
iteration 0 / 1000: loss 2.302774112771274
iteration 100 / 1000: loss 1.913518883359934
iteration 200 / 1000: loss 1.7662738696946416
iteration 300 / 1000: loss 1.6143280430411133
iteration 400 / 1000: loss 1.5172801735235812
iteration 500 / 1000: loss 1.5814246969408219
iteration 600 / 1000: loss 1.630279773696954
iteration 700 / 1000: loss 1.4597148063165972
iteration 800 / 1000: loss 1.4185925677272049
iteration 900 / 1000: loss 1.5220000988029048
Validation accuracy:  0.458 for learning rate= 0.0014 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027759664938667
iteration 100 / 1000: loss 1.8502196236758943
iteration 200 / 1000: loss 1.6561324366184307
iteration 300 / 1000: loss 1.633037661366044
iteration 400 / 1000: loss 1.5621801150459207
iteration 500 / 1000: loss 1.555079668167577
iteration 600 / 1000: loss 1.442662989002445
iteration 700 / 1000: loss 1.5436268160716338
iteration 800 / 1000: loss 1.6140923532723623
iteration 900 / 1000: loss 1.6485802536415806
```

```
Validation accuracy:  0.486 for learning rate= 0.0015 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027831931343314
iteration 100 / 1000: loss 1.8799633369913613
iteration 200 / 1000: loss 1.7437969641831665
iteration 300 / 1000: loss 1.5934710322973362
iteration 400 / 1000: loss 1.643125191684595
iteration 500 / 1000: loss 1.4741327505926718
iteration 600 / 1000: loss 1.6807849502063543
iteration 700 / 1000: loss 1.5080744714033936
iteration 800 / 1000: loss 1.4904408322730713
iteration 900 / 1000: loss 1.5331369427690242
Validation accuracy:  0.502 for learning rate= 0.0016 and decay rate= 0.93
iteration 0 / 1000: loss 2.302776308975928
iteration 100 / 1000: loss 1.8386854484027673
iteration 200 / 1000: loss 1.7380583349812813
iteration 300 / 1000: loss 1.6575012337082617
iteration 400 / 1000: loss 1.5826972138729605
iteration 500 / 1000: loss 1.5097910113000492
iteration 600 / 1000: loss 1.422820614007086
iteration 700 / 1000: loss 1.6510415941271193
iteration 800 / 1000: loss 1.5023397592744636
iteration 900 / 1000: loss 1.4820162337293827
Validation accuracy:  0.466 for learning rate= 0.0018 and decay rate= 0.93
iteration 0 / 1000: loss 2.3027580897821105
iteration 100 / 1000: loss 1.8134832652077677
iteration 200 / 1000: loss 1.673572618171472
iteration 300 / 1000: loss 1.514317595263018
iteration 400 / 1000: loss 1.6859639083136124
iteration 500 / 1000: loss 1.4915345097951742
iteration 600 / 1000: loss 1.4577402104140278
iteration 700 / 1000: loss 1.680291860404149
iteration 800 / 1000: loss 1.7381266018848855
iteration 900 / 1000: loss 1.547767205001409
Validation accuracy:  0.451 for learning rate= 0.002 and decay rate= 0.93
iteration 0 / 1000: loss 2.3028006912047427
iteration 100 / 1000: loss 2.3024373819498543
iteration 200 / 1000: loss 2.3004167287282278
iteration 300 / 1000: loss 2.280025715281056
iteration 400 / 1000: loss 2.207597840718256
iteration 500 / 1000: loss 2.1105841798573826
iteration 600 / 1000: loss 2.1408763274717715
iteration 700 / 1000: loss 2.0620071802265523
iteration 800 / 1000: loss 2.057689566109094
iteration 900 / 1000: loss 2.0125096320131033
Validation accuracy:  0.277 for learning rate= 0.0001 and decay rate= 0.93
best validation accuracy  0.502 achieved for learning rate 0.0016 and decay 0.93
```

[31]:
```
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)
```





0.7 Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

0.8 Answer:

(1) It is obvious that the subopt net has figured out some rough template features of classes. Eg we roughly see the outline of a car in some entries of the grip plot. Whoever, the features detectd by teh subopt are so rough that they could only lead to a bad performance. On the contrary, in the best network achieved, we see that the template features detected are much more detailed, and naturally best_net is able to classify photos much more accurately.

0.9 Evaluate on test set

[34]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.469
```

[ ]:

[ ]: