two_layer_nn

February 9, 2024

0.1 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.

[ ]:
```
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))))
```

0.2 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

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

[ ]:
```
# 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

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

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231214460989e-08
```

0.2.2 Forward pass loss

[ ]:
```
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
```

0.2.3 Backward pass

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

[ ]:
```
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[param_name])))
```

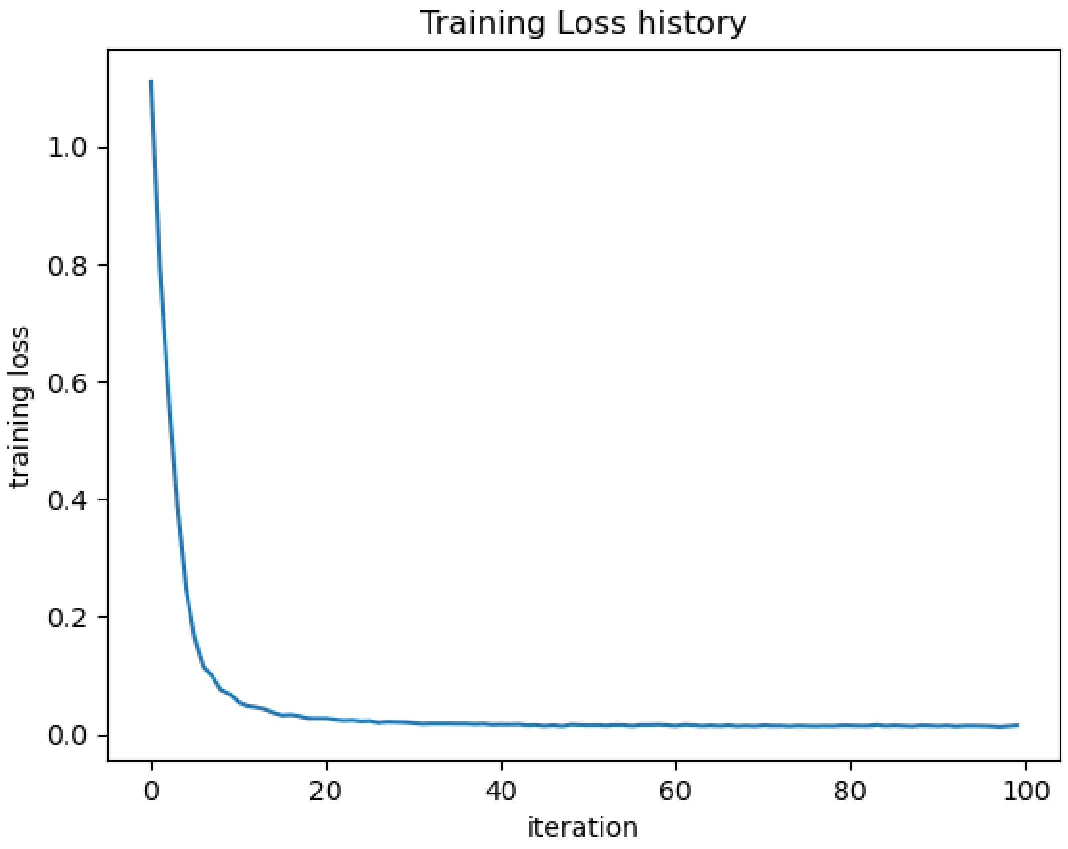
```
b2 max relative error: 1.248270530283678e-09
W2 max relative error: 2.9632227682005116e-10
b1 max relative error: 3.172680092703762e-09
W1 max relative error: 1.2832823337649917e-09
```

0.2.4 Training the network

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

```
[ ]: net = init_toy_model()
     stats = net.train(X, y, X, y,
                 learning_rate=1e-1, reg=5e-6,
                 num_iters=100, verbose=False)

     print('Final training loss: ', stats['loss_history'][-1])

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

```
Final training loss:  0.014497864587765886
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```
[ ]: 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' # remember to use correct path
    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%.

[ ]:
```
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
```

0.4 Questions:

The training accuracy isn't great.

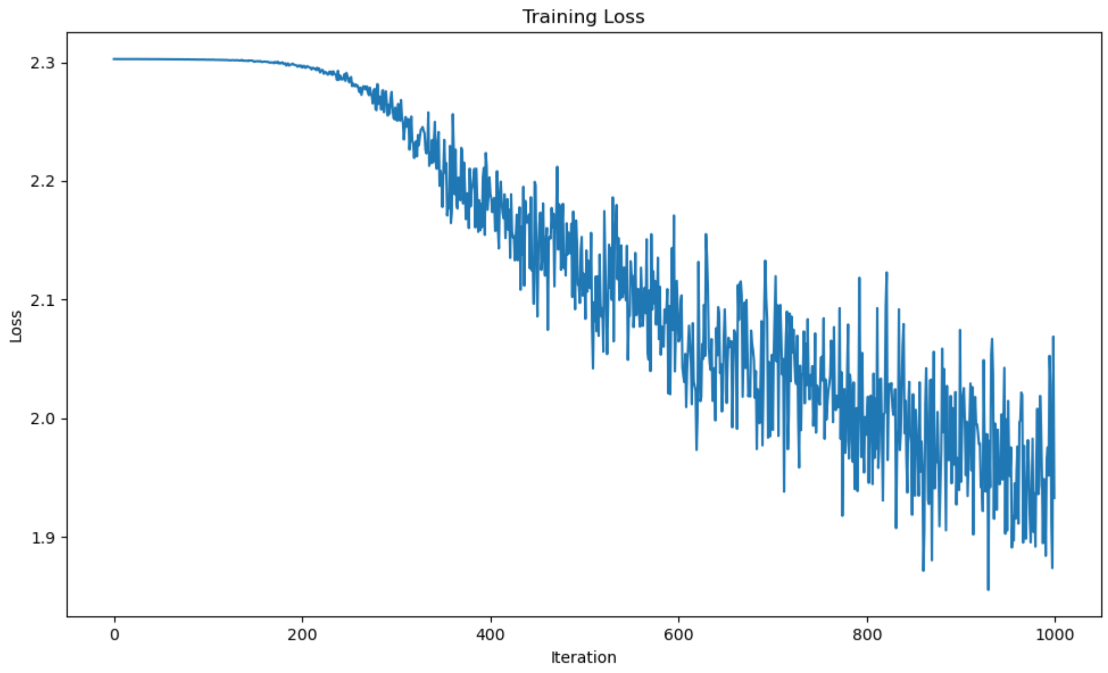
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

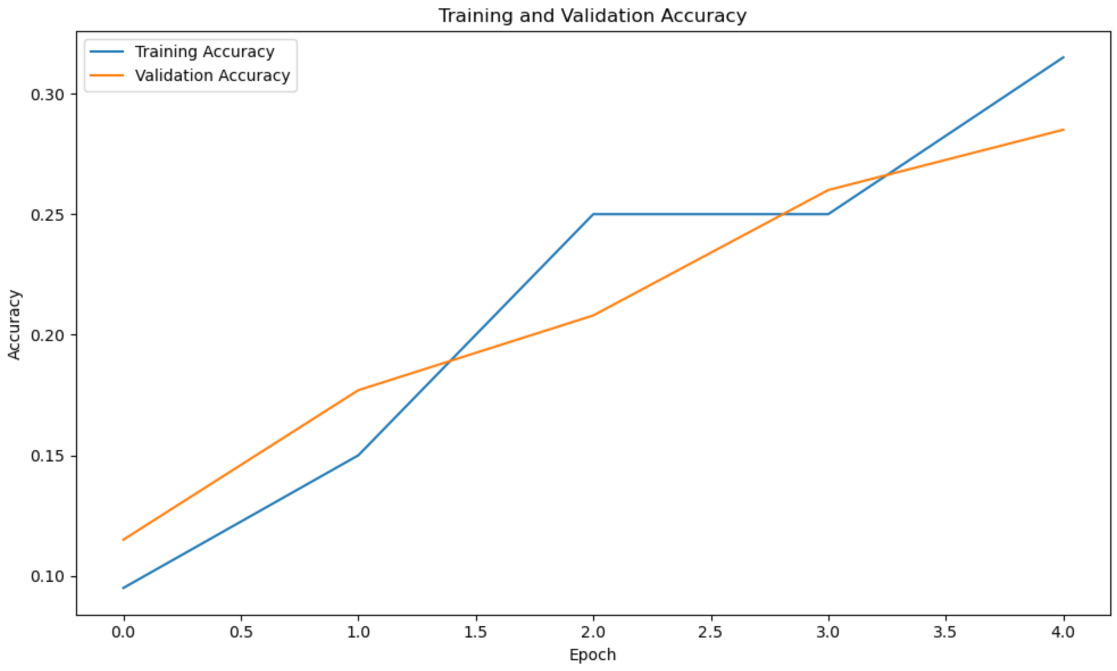
(2) How should you fix the problems you identified in (1)?

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

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

```
[ ]: # ================================================================ #
     # YOUR CODE HERE:
     #   Do some debugging to gain some insight into why the optimization
     #   isn't great.
     # ================================================================ #

     # Plot the loss function and train / validation accuracies
     plt.figure(figsize=(10,12))

     # Plot loss
     plt.subplot(2, 1, 1)
     plt.plot(stats['loss_history'], label='Training Loss')
     plt.xlabel('Iteration')
     plt.ylabel('Loss')
     plt.title('Training Loss')

     # Plot training and validation accuracies
     plt.subplot(2, 1, 2)
     plt.plot(stats['train_acc_history'], label='Training Accuracy')
     plt.plot(stats['val_acc_history'], label='Validation Accuracy')
     plt.xlabel('Epoch')
     plt.ylabel('Accuracy')
     plt.title('Training and Validation Accuracy')
     plt.legend()

     # Show plot
     plt.tight_layout()
     plt.show()


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





0.5 Answers:

(1) There are a few reasons the training accuracy could be underperforming. For one, we can see by the plots above that the validation & training accuracies are still improving at the end of the fourth epoch. This suggests that the model has not had enough training epochs to converge on the best model (the loss is still declining and has not yet converged either after the selected number of iterations). This is a sign of underfitting. Additionally, it's always

worth optimizing other hyperparameters to find the best model (num iterations, batch size, learning rate, etc.). These can all have a huge impact on performance.

(2) As stated above, we should explore a longer training sequence (increase num epochs) and we should also play around with optimizing some of the hyperparameters. If we increase the number of epochs, it's usually a good idea to lower the learning rate to avoid overfitting.

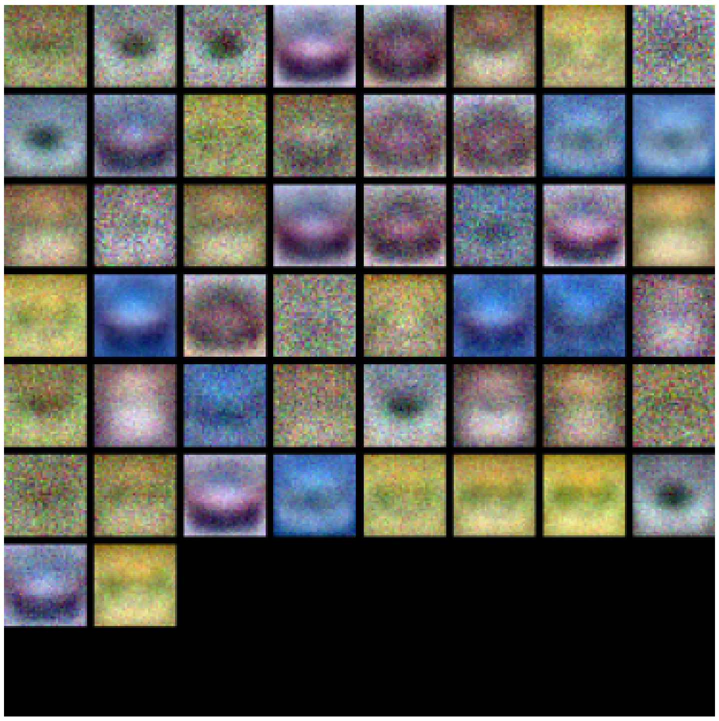
0.6 Optimize the neural network

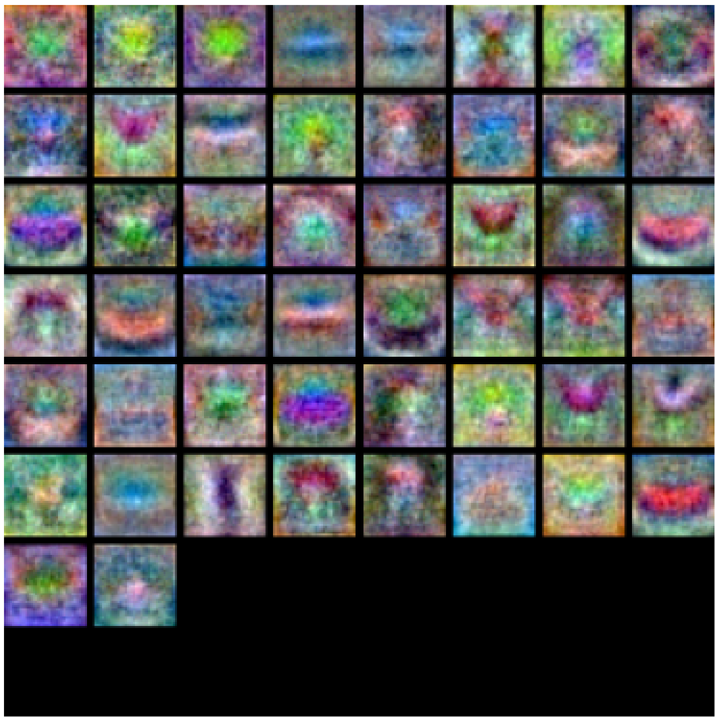
Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

```
[ ]: 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
     best_net = TwoLayerNet(input_size, hidden_size, num_classes)

     # Train the network
     stats = best_net.train(X_train, y_train, X_val, y_val,
                     num_iters=10000,
                     batch_size=200,
                     learning_rate=2e-4,
                     learning_rate_decay=0.95,
                     reg=0.2,
                     verbose=True)

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

```
iteration 0 / 10000: loss 2.3027409020971383
iteration 100 / 10000: loss 2.298690755501941
iteration 200 / 10000: loss 2.1782721624646006
iteration 300 / 10000: loss 2.0508064745941
iteration 400 / 10000: loss 2.049424906812619
iteration 500 / 10000: loss 1.8946120601817058
iteration 600 / 10000: loss 1.912604823348418
iteration 700 / 10000: loss 1.8218921609668284
iteration 800 / 10000: loss 1.8261714391140975
iteration 900 / 10000: loss 1.7966296722495816
iteration 1000 / 10000: loss 1.8152419001132323
iteration 1100 / 10000: loss 1.7836799785949908
iteration 1200 / 10000: loss 1.6364268999348568
iteration 1300 / 10000: loss 1.6796948381041656
iteration 1400 / 10000: loss 1.6718446373151157
iteration 1500 / 10000: loss 1.5275104199403982
iteration 1600 / 10000: loss 1.680576007948543
iteration 1700 / 10000: loss 1.801166927936333
iteration 1800 / 10000: loss 1.6255510963534112
iteration 1900 / 10000: loss 1.6585424924753946
iteration 2000 / 10000: loss 1.714559082479604
iteration 2100 / 10000: loss 1.4361975367030102
iteration 2200 / 10000: loss 1.6438218493759573
iteration 2300 / 10000: loss 1.557211061787521
iteration 2400 / 10000: loss 1.6627186960374862
iteration 2500 / 10000: loss 1.6773009707626518
iteration 2600 / 10000: loss 1.591748515622689
iteration 2700 / 10000: loss 1.6223002344267958
iteration 2800 / 10000: loss 1.5616691733318355
iteration 2900 / 10000: loss 1.5826298538639574
iteration 3000 / 10000: loss 1.4993727387727973
iteration 3100 / 10000: loss 1.4983730192410862
iteration 3200 / 10000: loss 1.6843528594409916
iteration 3300 / 10000: loss 1.5732881778207344
iteration 3400 / 10000: loss 1.5840828820778345
iteration 3500 / 10000: loss 1.5405200097703615
iteration 3600 / 10000: loss 1.5983183733225883
iteration 3700 / 10000: loss 1.6983089308822266
iteration 3800 / 10000: loss 1.5552462328357488
iteration 3900 / 10000: loss 1.438798143628768
iteration 4000 / 10000: loss 1.4496164594280572
iteration 4100 / 10000: loss 1.5141689582906714
iteration 4200 / 10000: loss 1.5400721589319513
iteration 4300 / 10000: loss 1.5452315596485093
iteration 4400 / 10000: loss 1.5118479633122268
iteration 4500 / 10000: loss 1.5342374782267076
iteration 4600 / 10000: loss 1.5029081186244633
iteration 4700 / 10000: loss 1.4400738690049775
iteration 4800 / 10000: loss 1.4457449212680564
iteration 4900 / 10000: loss 1.5354604604576654
iteration 5000 / 10000: loss 1.5506260269887209
iteration 5100 / 10000: loss 1.3846215655913832
iteration 5200 / 10000: loss 1.3283956219308029
iteration 5300 / 10000: loss 1.3903534385390834
iteration 5400 / 10000: loss 1.5211674060693212
iteration 5500 / 10000: loss 1.3215302225189511
iteration 5600 / 10000: loss 1.473636621632136
iteration 5700 / 10000: loss 1.4966213928631213
iteration 5800 / 10000: loss 1.508301540500292
iteration 5900 / 10000: loss 1.5766976298945012
iteration 6000 / 10000: loss 1.4434425595514135
iteration 6100 / 10000: loss 1.470577603427074
iteration 6200 / 10000: loss 1.4479529012204897
iteration 6300 / 10000: loss 1.5132532012144209
iteration 6400 / 10000: loss 1.5077309526745137
iteration 6500 / 10000: loss 1.455703149841408
iteration 6600 / 10000: loss 1.3830612442358132
iteration 6700 / 10000: loss 1.4564621664568214
iteration 6800 / 10000: loss 1.3763751230420946
iteration 6900 / 10000: loss 1.5304592161169697
iteration 7000 / 10000: loss 1.471867678413251
iteration 7100 / 10000: loss 1.5383800114627992
iteration 7200 / 10000: loss 1.5815282821364183
iteration 7300 / 10000: loss 1.3674483489468077
iteration 7400 / 10000: loss 1.4532334445179291
iteration 7500 / 10000: loss 1.441846822373098
iteration 7600 / 10000: loss 1.2939515752136765
iteration 7700 / 10000: loss 1.5000864409602477
iteration 7800 / 10000: loss 1.5648395057546267
iteration 7900 / 10000: loss 1.3918529575282397
iteration 8000 / 10000: loss 1.3958082812514203
iteration 8100 / 10000: loss 1.4011133650019125
iteration 8200 / 10000: loss 1.443148479198828
iteration 8300 / 10000: loss 1.3869537442836253
iteration 8400 / 10000: loss 1.404134517140169
iteration 8500 / 10000: loss 1.332361252699819
iteration 8600 / 10000: loss 1.231533204645676
iteration 8700 / 10000: loss 1.560537390222309
iteration 8800 / 10000: loss 1.5987398311145031
iteration 8900 / 10000: loss 1.5720019794370095
iteration 9000 / 10000: loss 1.302487254965138
iteration 9100 / 10000: loss 1.3793550391285434
iteration 9200 / 10000: loss 1.4360691671569477
iteration 9300 / 10000: loss 1.497967665956823
iteration 9400 / 10000: loss 1.3702331940366308
iteration 9500 / 10000: loss 1.5666587100323577
iteration 9600 / 10000: loss 1.4383512101868654
iteration 9700 / 10000: loss 1.3676469792156434
iteration 9800 / 10000: loss 1.4547916820023645
iteration 9900 / 10000: loss 1.3950466632612741
Validation accuracy:  0.501
```

```
[ ]: 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)
```





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) The suboptimal weights appear to be much more generalized and “over-simplified”, whereas the optimized net has much more refined, complex weights that appear to better capture the nuances in the feature set.

0.9 Evaluate on test set

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

```
Test accuracy:  0.478
```