two_layer_nn

February 6, 2023

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.

[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))))
```

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

[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.
## 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:
3.3812312026648694e-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("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.

[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[param_name])))
```

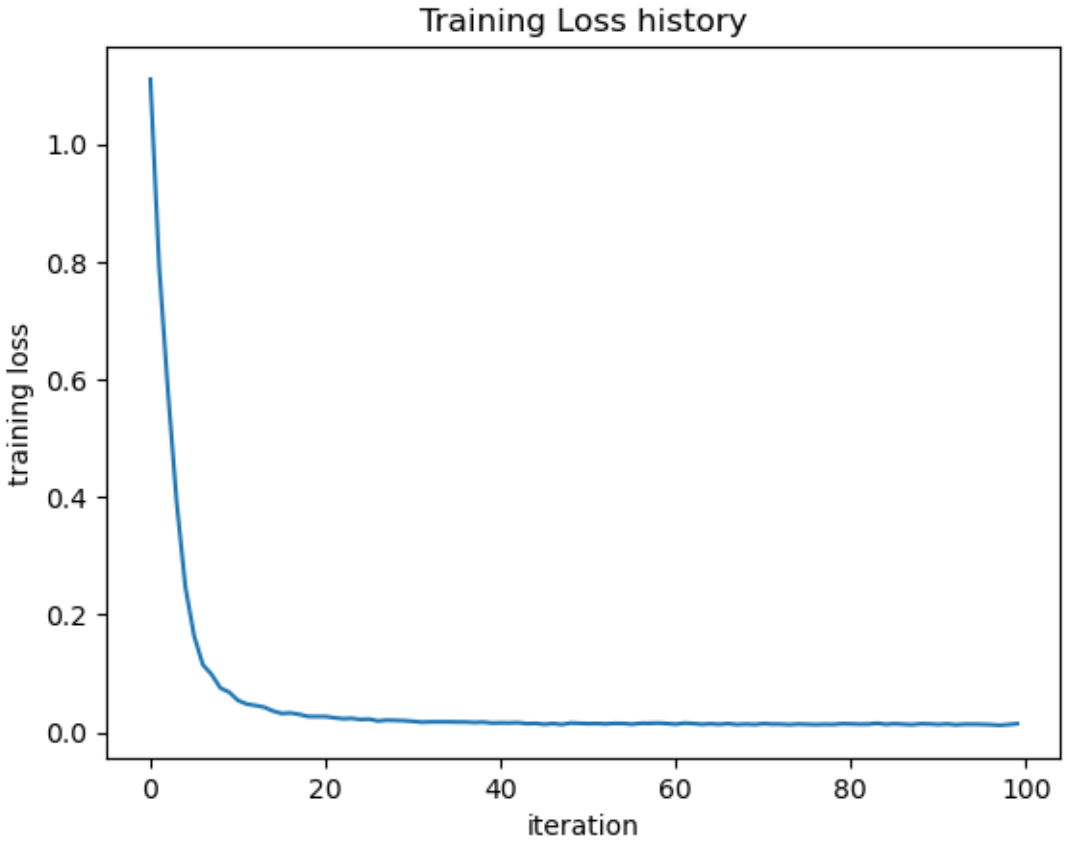
```
W2 max relative error: 2.9632221903873815e-10
b2 max relative error: 1.2482624742512528e-09
W1 max relative error: 1.283285096965795e-09
b1 max relative error: 3.172680285697327e-09
```

0.2.4 Training the network

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

[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:  0.014497864587765906
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[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,)
```

0.3.1 Running SGD

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

[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 1.9465176817856495
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)?

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

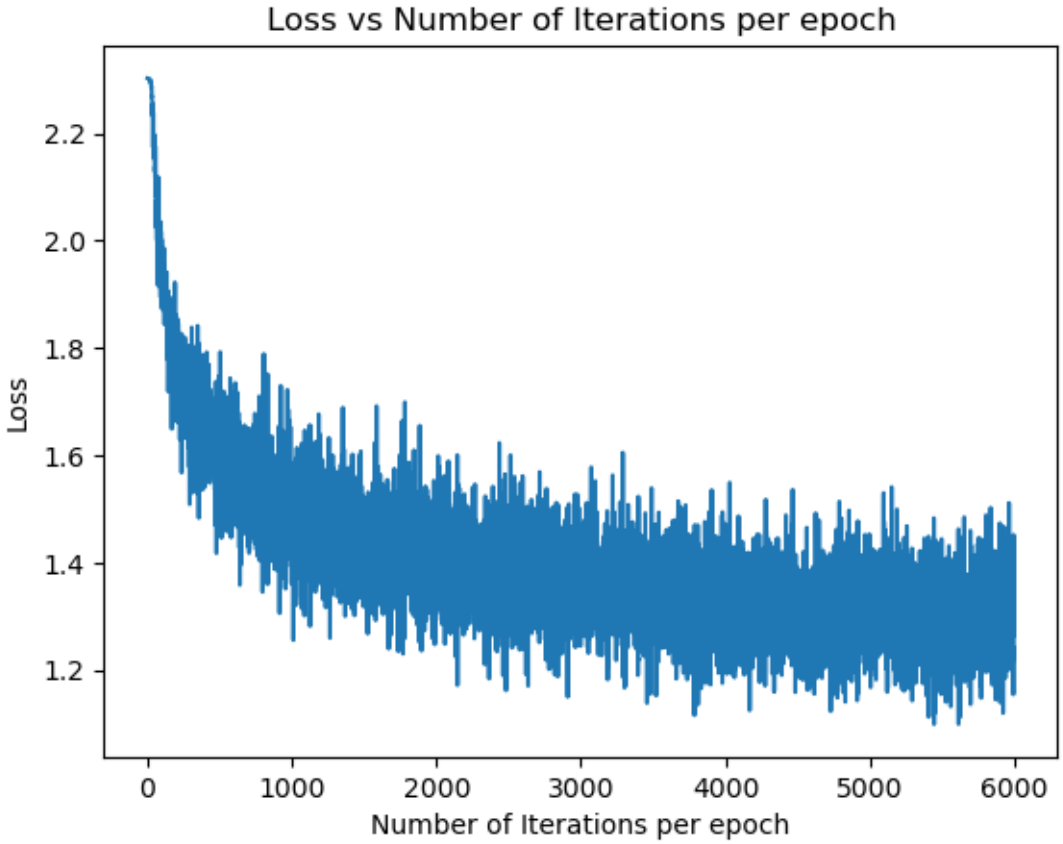
[10]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

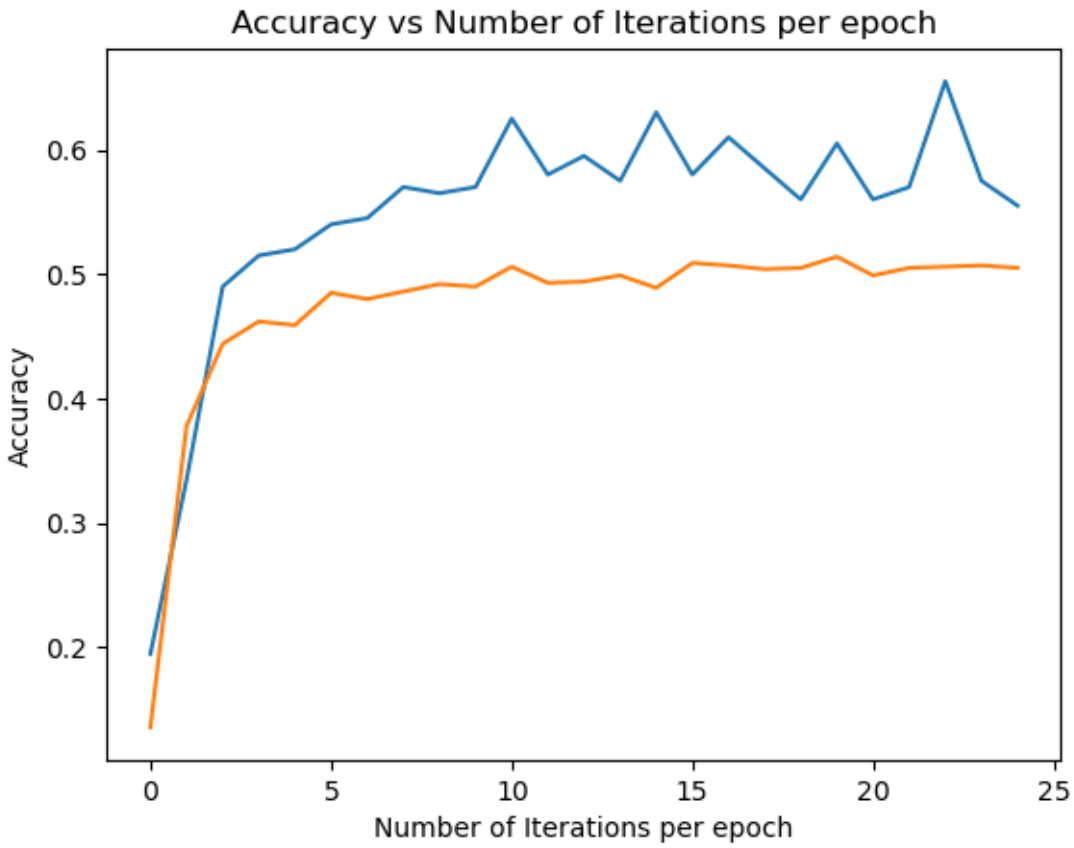
[11]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies
#Repeat the SGD Step with different parameters
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=6000, batch_size=200,
            learning_rate=1e-3, learning_rate_decay=0.90,
            reg=0.25, verbose=True)

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

loss_hist = stats['loss_history']
train_acc_hist = stats['train_acc_history']
val_acc_hist = stats['val_acc_history']

plt.figure()
plt.plot(loss_hist)
plt.ylabel('Loss')
plt.xlabel('Number of Iterations per epoch')
plt.title('Loss vs Number of Iterations per epoch')

plt.figure()
plt.plot(train_acc_hist, label = "train accuracy")
plt.plot(val_acc_hist, label = "validation accuracy")
plt.ylabel("Accuracy")
plt.xlabel("Number of Iterations per epoch")
plt.title("Accuracy vs Number of Iterations per epoch")
plt.show()

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

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

```
iteration 0 / 6000: loss 2.3027667167979295
iteration 100 / 6000: loss 1.9404733316030036
iteration 200 / 6000: loss 1.748953146006704
iteration 300 / 6000: loss 1.7916153234883794
iteration 400 / 6000: loss 1.6458612866036604
iteration 500 / 6000: loss 1.7402633280858502
iteration 600 / 6000: loss 1.5777918936535436
iteration 700 / 6000: loss 1.5751687419334839
iteration 800 / 6000: loss 1.5286121675845241
iteration 900 / 6000: loss 1.5991622956610243
iteration 1000 / 6000: loss 1.4704055302653272
iteration 1100 / 6000: loss 1.4466740620083518
iteration 1200 / 6000: loss 1.4445746439262066
iteration 1300 / 6000: loss 1.496975257421716
iteration 1400 / 6000: loss 1.4671836666518823
iteration 1500 / 6000: loss 1.4042933729695422
iteration 1600 / 6000: loss 1.4186221293573746
iteration 1700 / 6000: loss 1.4837463924515517
iteration 1800 / 6000: loss 1.3472188706315646
iteration 1900 / 6000: loss 1.518001964863528
iteration 2000 / 6000: loss 1.3921498797887422
iteration 2100 / 6000: loss 1.3278225915525204
iteration 2200 / 6000: loss 1.3540965029369607
iteration 2300 / 6000: loss 1.430682728587736
iteration 2400 / 6000: loss 1.297643589626398
iteration 2500 / 6000: loss 1.413752965792457
iteration 2600 / 6000: loss 1.4834302783376967
iteration 2700 / 6000: loss 1.4363693541093578
iteration 2800 / 6000: loss 1.3593306798738034
iteration 2900 / 6000: loss 1.2989356929347975
iteration 3000 / 6000: loss 1.3167163894891563
iteration 3100 / 6000: loss 1.5521004163931775
iteration 3200 / 6000: loss 1.3458851717819205
iteration 3300 / 6000: loss 1.2859104778360761
iteration 3400 / 6000: loss 1.415604959788035
iteration 3500 / 6000: loss 1.4291855574588002
iteration 3600 / 6000: loss 1.3687859225168144
iteration 3700 / 6000: loss 1.2794976739282848
iteration 3800 / 6000: loss 1.3329578284256118
iteration 3900 / 6000: loss 1.369104853615233
iteration 4000 / 6000: loss 1.3616150720852762
iteration 4100 / 6000: loss 1.2569525174196374
iteration 4200 / 6000: loss 1.3113042840087268
iteration 4300 / 6000: loss 1.2508031795879733
iteration 4400 / 6000: loss 1.321551638596598
iteration 4500 / 6000: loss 1.3137962446338949
```

```
iteration 4600 / 6000: loss 1.2790773942221902
iteration 4700 / 6000: loss 1.2906263336618207
iteration 4800 / 6000: loss 1.3992115818725548
iteration 4900 / 6000: loss 1.3192905331528262
iteration 5000 / 6000: loss 1.30938229032305
iteration 5100 / 6000: loss 1.4358646614154074
iteration 5200 / 6000: loss 1.2204546535600407
iteration 5300 / 6000: loss 1.3533830876401554
iteration 5400 / 6000: loss 1.2475882073096558
iteration 5500 / 6000: loss 1.2522153471491868
iteration 5600 / 6000: loss 1.3417726614261052
iteration 5700 / 6000: loss 1.3244879281216737
iteration 5800 / 6000: loss 1.4227985828128487
iteration 5900 / 6000: loss 1.4734598355935975
Validation accuracy:  0.512
```





0.5 Answers:

(1) The main reason behind low validation accuracy is that learning stops early. Therefore, model underfits. There are possible reasons why the model underfits. Firstly, learning_rate is too small which causes model to not reach to minimum loss. Second reason is the number of iterations. Number of iterations should be increased to avoid early stopping. Thirdly, learning_rate_decay is too high which causes learning_rate to get small very quickly which leads to the first problem. In brief, grid search of hyperparameters are required to obtain good validation accuracy.

(2) I have increased number of iterations and learning rate to increase validation accuracy to 51.2%. Also, I have decreased learning_rate_decay to 0.9 from 0.95 which also increased validation accuracy.

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.

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

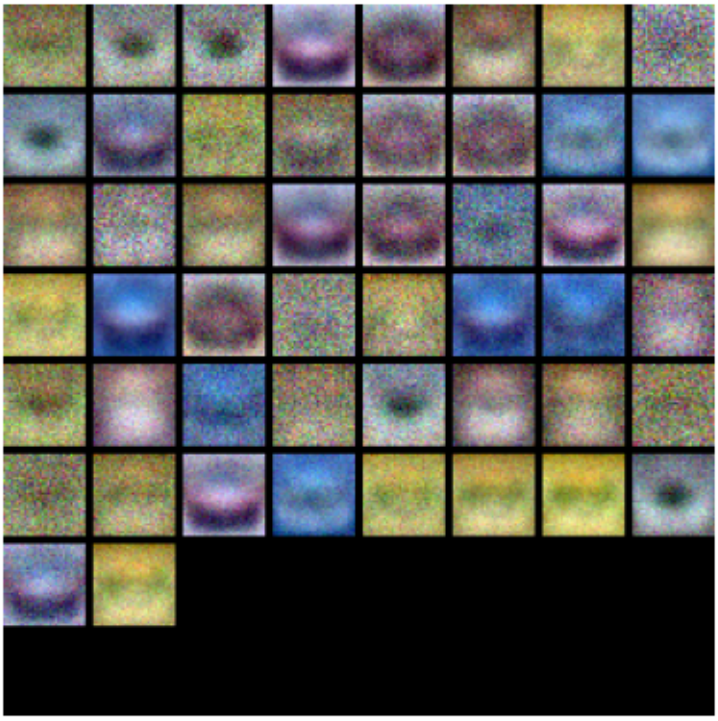
```
# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   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

lr_list = [1e-3]
reg_list = [0.05,0.15,0.25]
iter_list = [5000]
learning_rate_decay_list = [0.85,0.9,0.95]
batch_size_list = [200]
best_val = 0
for num_iter in iter_list:
    for lr in lr_list:
        for rg in reg_list:
            for lr_decay in learning_rate_decay_list:
                for batch_sz in batch_size_list:
                    net = TwoLayerNet(input_size, hidden_size, num_classes)

                    # Train the network

                    stats = net.train(X_train, y_train, X_val, y_val,
                            num_iters=num_iter, batch_size=batch_sz,
                            learning_rate=lr, learning_rate_decay=lr_decay,
                            reg=rg, verbose=False)

                    # Predict on the validation set
                    val_acc = (net.predict(X_val) == y_val).mean()
                    print(f'num_iter ={num_iter},lr={lr}, reg={rg},␣
 ↪lr_decay={lr_decay},batch_sz ={batch_sz},val_acc={val_acc}')
                    if val_acc > best_val:
                          best_val = val_acc
                          best_net = net
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
```

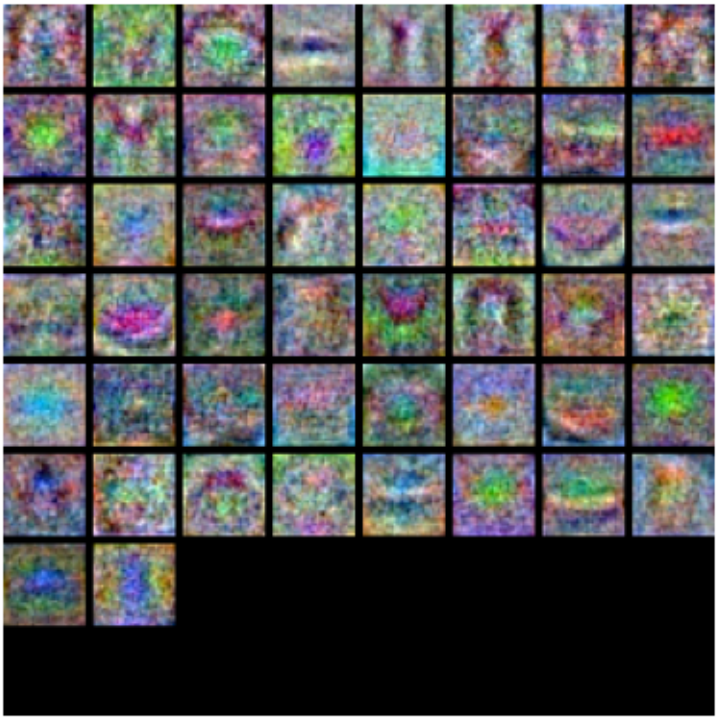
```
print('Validation accuracy: ', val_acc)
```

```
num_iter =5000,lr=0.001, reg=0.05, lr_decay=0.85,batch_sz =200,val_acc=0.502
num_iter =5000,lr=0.001, reg=0.05, lr_decay=0.9,batch_sz =200,val_acc=0.501
num_iter =5000,lr=0.001, reg=0.05, lr_decay=0.95,batch_sz =200,val_acc=0.523
num_iter =5000,lr=0.001, reg=0.15, lr_decay=0.85,batch_sz =200,val_acc=0.493
num_iter =5000,lr=0.001, reg=0.15, lr_decay=0.9,batch_sz =200,val_acc=0.538
num_iter =5000,lr=0.001, reg=0.15, lr_decay=0.95,batch_sz =200,val_acc=0.52
num_iter =5000,lr=0.001, reg=0.25, lr_decay=0.85,batch_sz =200,val_acc=0.496
num_iter =5000,lr=0.001, reg=0.25, lr_decay=0.9,batch_sz =200,val_acc=0.513
num_iter =5000,lr=0.001, reg=0.25, lr_decay=0.95,batch_sz =200,val_acc=0.535
Validation accuracy:  0.538
```

Best net parameters are: num_iter =5000,lr=0.001, reg=0.15, lr_decay=0.9,batch_sz =200,val_acc=0.538

[14]:
```
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) Weights of suboptimal nets are very similar to each other which indicates that learning is not complete. However, weights of best net have distinct and distinguishable shapes which indicates that neural network was able to extract distinct features of images.

0.9 Evaluate on test set

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

```
Test accuracy:  0.514
```

[ ]:
```

```