two_layer_net

May 8, 2018

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

```
In [24]: # A bit of setup
         # Reference: https://github.com/Halfish/cs231n/tree/master/assignment1

         import numpy as np
         import matplotlib.pyplot as plt

         from hmwk5_2.classifiers.neural_net import TwoLayerNet

         from __future__ import print_function

         %matplotlib inline
         plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
         plt.rcParams['image.interpolation'] = 'nearest'
         plt.rcParams['image.cmap'] = 'gray'

         # for auto-reloading external modules
         # see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
         %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))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

We will use the class `TwoLayerNet` in the file `hmwk5_2/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

```
In [25]: # 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()
```

2 Forward pass: compute scores

Open the file `hmwk5_2/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

```
In [26]: scores = net.loss(X)
         print('Your scores:')
         print(scores)
         print()
         print('correct scores:')
         correct_scores = np.asarray([
           [-0.81233741, -1.27654624, -0.70335995],
           [-0.17129677, -1.18803311, -0.47310444],
           [-0.51590475, -1.01354314, -0.8504215 ],
           [-0.15419291, -0.48629638, -0.52901952],
           [-0.00618733, -0.12435261, -0.15226949]])
         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:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720710348014e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

```
In [27]: loss, _ = net.loss(X, y, reg=0.05)
         correct_loss = 1.30378789133

         # 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:
0.018965419606063127
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
In [28]: from hmwk5_2.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('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 1.276034e-10
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
```

5 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

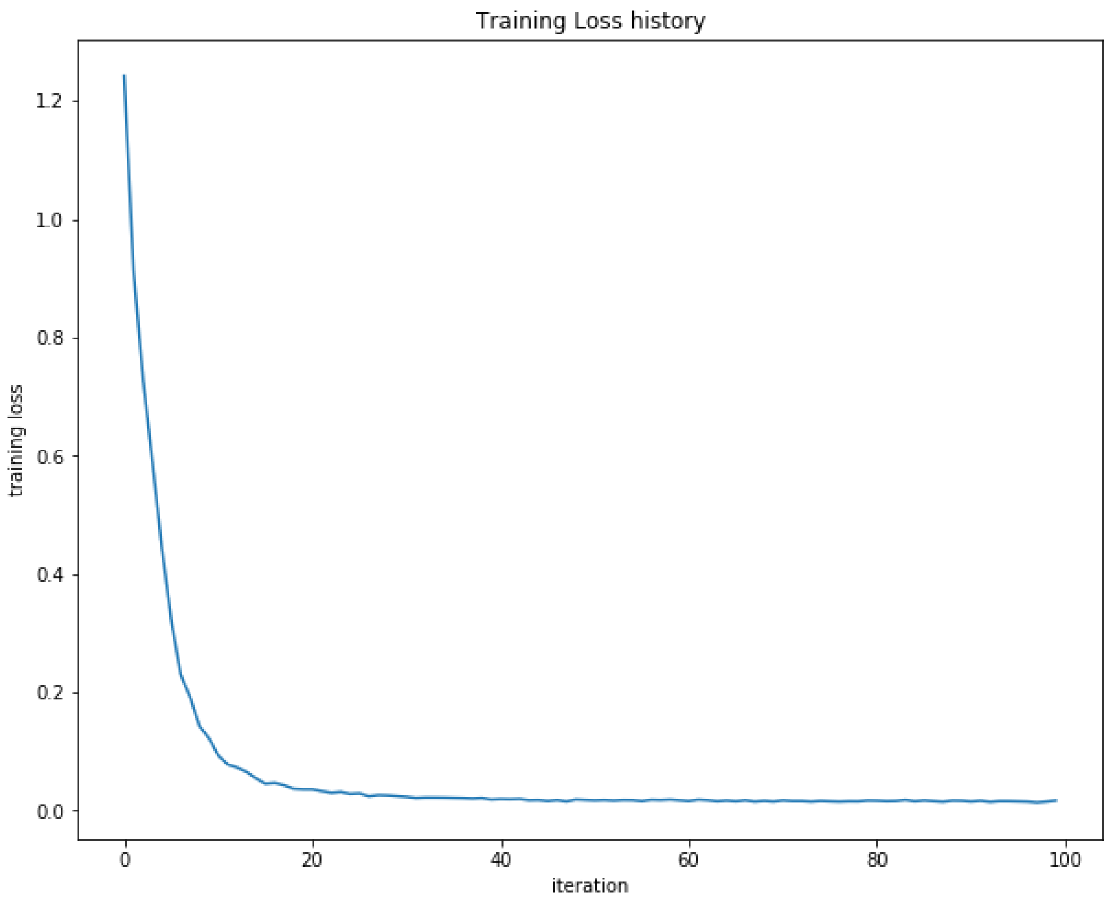
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

```
In [29]: 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:  0.017143645815455258
```



6 Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

```
In [30]: from hmwk5_2.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 = 'hmwk5_2/datasets/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


    # Cleaning up variables to prevent loading data multiple times (which may cause memor
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    # 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)
```

```
Clear previously loaded data.
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,)
```

7 Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

```
In [31]: 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)
```

```
iteration 0 / 1000: loss 2.302762
iteration 100 / 1000: loss 2.302358
iteration 200 / 1000: loss 2.297405
iteration 300 / 1000: loss 2.258897
iteration 400 / 1000: loss 2.202976
iteration 500 / 1000: loss 2.116818
iteration 600 / 1000: loss 2.049801
iteration 700 / 1000: loss 1.985725
iteration 800 / 1000: loss 2.003727
iteration 900 / 1000: loss 1.948067
Validation accuracy:  0.287
```

8 Debug the training

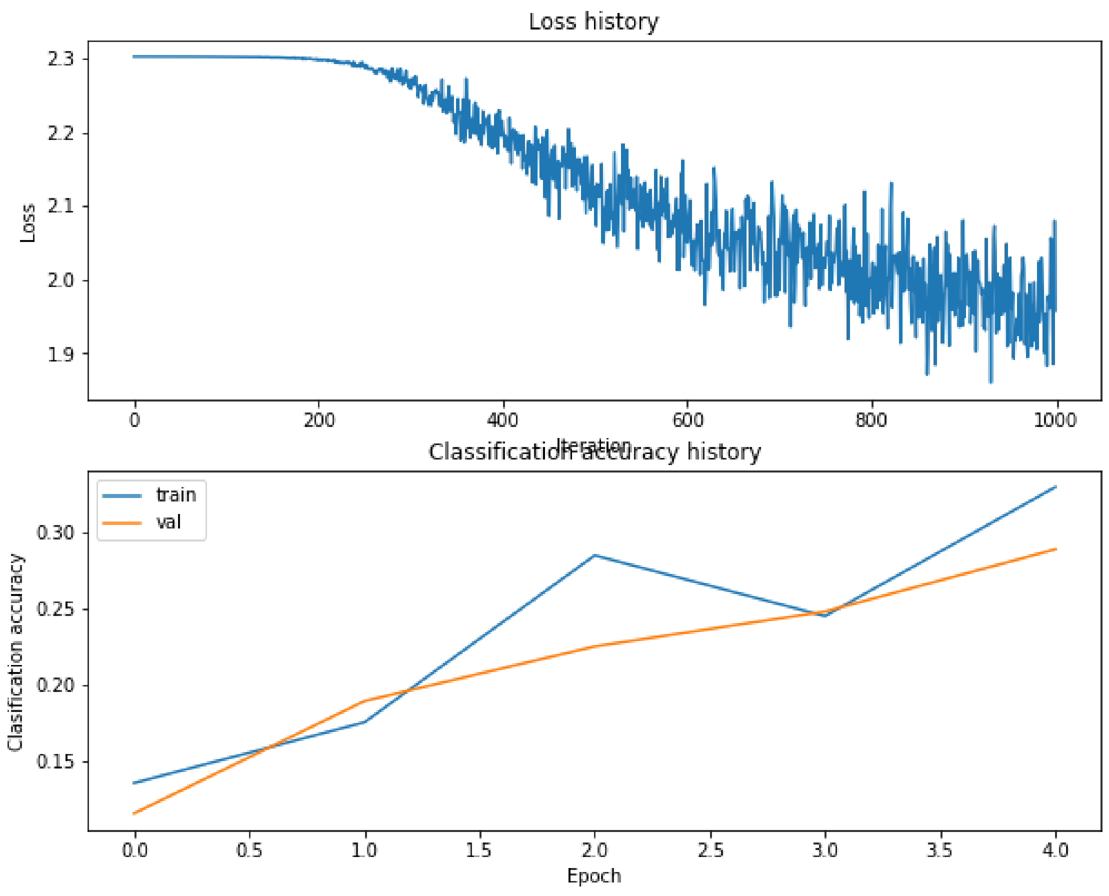
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

```
In [32]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(stats['loss_history'])
         plt.title('Loss history')
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         plt.subplot(2, 1, 2)
         plt.plot(stats['train_acc_history'], label='train')
         plt.plot(stats['val_acc_history'], label='val')
         plt.title('Classification accuracy history')
         plt.xlabel('Epoch')
         plt.ylabel('Clasification accuracy')
         plt.legend()
         plt.show()
```

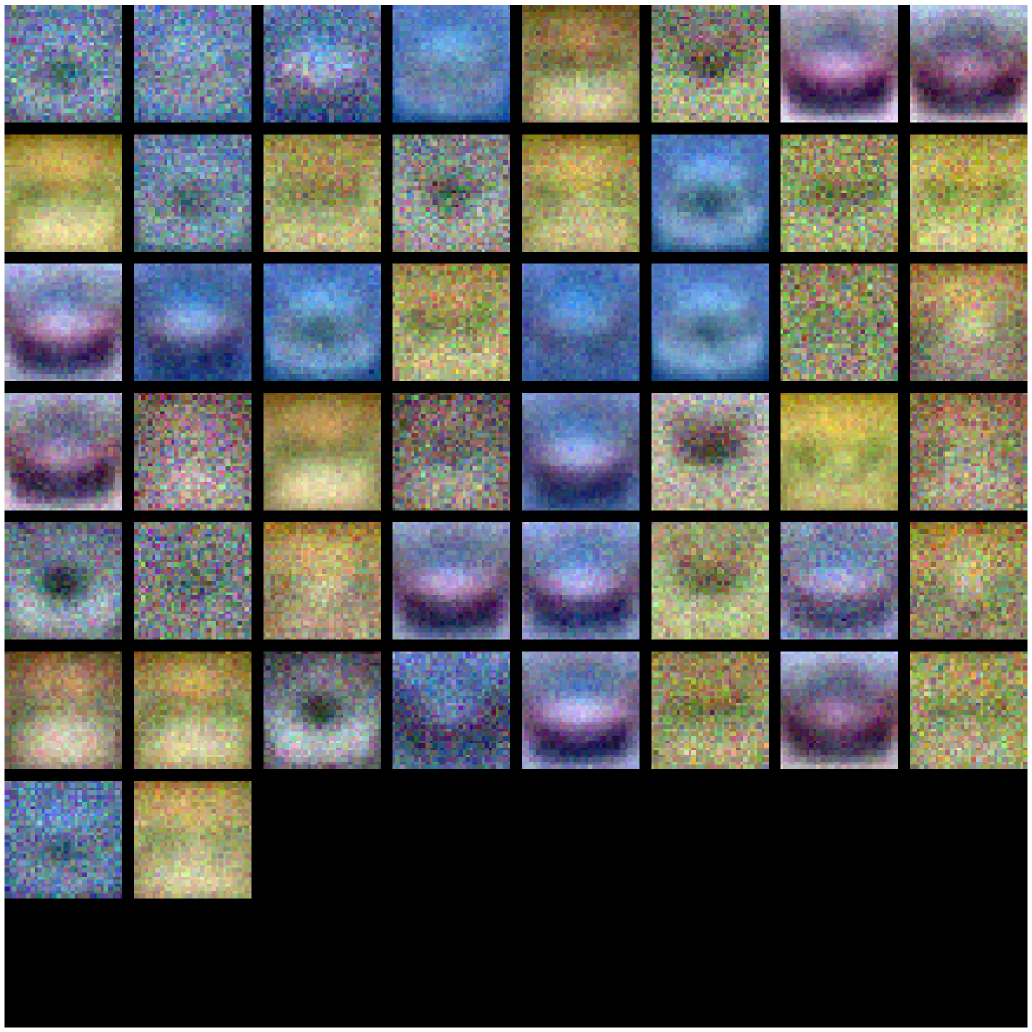


```
In [33]: from hmwk5_2.vis_utils import visualize_grid

         # Visualize the weights of the network
```

```
def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.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(net)
```



9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

```
In [34]: best_net = None # store the best model into this

         #################################################################################
         # TODO: Tune hyperparameters using the validation set. Store your best trained  #
         # model in best_net.                                                            #
         #                                                                               #
         # To help debug your network, it may help to use visualizations similar to the  #
         # ones we used above; these visualizations will have significant qualitative    #
         # differences from the ones we saw above for the poorly tuned network.          #
         #                                                                               #
         # Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
         # write code to sweep through possible combinations of hyperparameters          #
         # automatically like we did on the previous exercises.                          #
         #################################################################################
         pass
         #batch_size = 400, lr = 0.001000, hidden size = 200.000000, Valid_accuracy: 0.518000#
         best_accuracy = 0
         input_size = 32 * 32 * 3
         num_classes = 10

         # Train the network with different parameters

         for bs in [200, 300, 400]:
             for lr in [1e-3, 1e-4, 1e-5]:
                 for hidden_size in [50, 100, 200]:
                     net = TwoLayerNet(input_size, hidden_size, num_classes)
                     stats = net.train(X_train, y_train, X_val, y_val,
                                 num_iters=1500, batch_size=bs,
                                 learning_rate=lr, learning_rate_decay=0.9,
                                 reg=0.5, verbose=True)

                     # Predict on the validation set
                     current_accuracy = (net.predict(X_val) == y_val).mean()
                     print ('batch_size = %d, lr = %f, hidden size = %f, Valid_accuracy: %f' %
                     if current_accuracy > best_accuracy:
                         best_accuracy = current_accuracy
                         best_net = net


         #best_net = TwoLayerNet(input_size, 200, 10)
         ################################################################################
         #                              END OF YOUR CODE                                #
         ################################################################################
```

```
iteration 0 / 1500: loss 2.302970
iteration 100 / 1500: loss 1.967770
iteration 200 / 1500: loss 1.761300
iteration 300 / 1500: loss 1.838307
iteration 400 / 1500: loss 1.682846
iteration 500 / 1500: loss 1.725149
iteration 600 / 1500: loss 1.609002
iteration 700 / 1500: loss 1.605850
iteration 800 / 1500: loss 1.533885
iteration 900 / 1500: loss 1.614207
iteration 1000 / 1500: loss 1.480595
iteration 1100 / 1500: loss 1.454163
iteration 1200 / 1500: loss 1.513307
iteration 1300 / 1500: loss 1.533797
iteration 1400 / 1500: loss 1.496025
batch_size = 200, lr = 0.001000, hidden size = 50.000000, Valid_accuracy: 0.484000
iteration 0 / 1500: loss 2.303327
iteration 100 / 1500: loss 1.957897
iteration 200 / 1500: loss 1.757293
iteration 300 / 1500: loss 1.634231
iteration 400 / 1500: loss 1.729293
iteration 500 / 1500: loss 1.568954
iteration 600 / 1500: loss 1.505275
iteration 700 / 1500: loss 1.620043
iteration 800 / 1500: loss 1.485448
iteration 900 / 1500: loss 1.528107
iteration 1000 / 1500: loss 1.418570
iteration 1100 / 1500: loss 1.491908
iteration 1200 / 1500: loss 1.511212
iteration 1300 / 1500: loss 1.453913
iteration 1400 / 1500: loss 1.559752
batch_size = 200, lr = 0.001000, hidden size = 100.000000, Valid_accuracy: 0.487000
```

```
iteration 0 / 1500: loss 2.304100
iteration 100 / 1500: loss 1.928187
iteration 200 / 1500: loss 1.673755
iteration 300 / 1500: loss 1.688768
iteration 400 / 1500: loss 1.578608
iteration 500 / 1500: loss 1.508357
iteration 600 / 1500: loss 1.604877
iteration 700 / 1500: loss 1.463824
iteration 800 / 1500: loss 1.517573
iteration 900 / 1500: loss 1.526169
iteration 1000 / 1500: loss 1.506388
iteration 1100 / 1500: loss 1.471880
iteration 1200 / 1500: loss 1.410452
iteration 1300 / 1500: loss 1.500435
iteration 1400 / 1500: loss 1.439676
batch_size = 200, lr = 0.001000, hidden size = 200.000000, Valid_accuracy: 0.496000
iteration 0 / 1500: loss 2.302960
iteration 100 / 1500: loss 2.302489
iteration 200 / 1500: loss 2.299310
iteration 300 / 1500: loss 2.271647
iteration 400 / 1500: loss 2.202846
iteration 500 / 1500: loss 2.116955
iteration 600 / 1500: loss 2.133877
iteration 700 / 1500: loss 2.033924
iteration 800 / 1500: loss 2.019419
iteration 900 / 1500: loss 2.032865
iteration 1000 / 1500: loss 2.068400
iteration 1100 / 1500: loss 1.993175
iteration 1200 / 1500: loss 1.987768
iteration 1300 / 1500: loss 1.964876
iteration 1400 / 1500: loss 1.962951
batch_size = 200, lr = 0.000100, hidden size = 50.000000, Valid_accuracy: 0.308000
iteration 0 / 1500: loss 2.303351
iteration 100 / 1500: loss 2.302017
iteration 200 / 1500: loss 2.291607
iteration 300 / 1500: loss 2.250005
iteration 400 / 1500: loss 2.198777
iteration 500 / 1500: loss 2.150425
iteration 600 / 1500: loss 2.029222
iteration 700 / 1500: loss 2.063255
iteration 800 / 1500: loss 2.037185
iteration 900 / 1500: loss 2.009670
iteration 1000 / 1500: loss 1.911266
iteration 1100 / 1500: loss 1.961985
iteration 1200 / 1500: loss 1.867283
iteration 1300 / 1500: loss 1.917448
iteration 1400 / 1500: loss 1.927229
batch_size = 200, lr = 0.000100, hidden size = 100.000000, Valid_accuracy: 0.308000
```

```
iteration 0 / 1500: loss 2.304097
iteration 100 / 1500: loss 2.302774
iteration 200 / 1500: loss 2.290965
iteration 300 / 1500: loss 2.264173
iteration 400 / 1500: loss 2.156672
iteration 500 / 1500: loss 2.033957
iteration 600 / 1500: loss 2.071679
iteration 700 / 1500: loss 2.039279
iteration 800 / 1500: loss 2.045696
iteration 900 / 1500: loss 2.033545
iteration 1000 / 1500: loss 1.956135
iteration 1100 / 1500: loss 1.993338
iteration 1200 / 1500: loss 1.886236
iteration 1300 / 1500: loss 1.888710
iteration 1400 / 1500: loss 1.907205
batch_size = 200, lr = 0.000100, hidden size = 200.000000, Valid_accuracy: 0.325000
iteration 0 / 1500: loss 2.302955
iteration 100 / 1500: loss 2.302938
iteration 200 / 1500: loss 2.302924
iteration 300 / 1500: loss 2.302913
iteration 400 / 1500: loss 2.302903
iteration 500 / 1500: loss 2.302829
iteration 600 / 1500: loss 2.302766
iteration 700 / 1500: loss 2.302750
iteration 800 / 1500: loss 2.302737
iteration 900 / 1500: loss 2.302682
iteration 1000 / 1500: loss 2.302611
iteration 1100 / 1500: loss 2.302580
iteration 1200 / 1500: loss 2.302491
iteration 1300 / 1500: loss 2.302417
iteration 1400 / 1500: loss 2.302250
batch_size = 200, lr = 0.000010, hidden size = 50.000000, Valid_accuracy: 0.204000
iteration 0 / 1500: loss 2.303322
iteration 100 / 1500: loss 2.303298
iteration 200 / 1500: loss 2.303243
iteration 300 / 1500: loss 2.303185
iteration 400 / 1500: loss 2.303129
iteration 500 / 1500: loss 2.303076
iteration 600 / 1500: loss 2.302959
iteration 700 / 1500: loss 2.302854
iteration 800 / 1500: loss 2.302814
iteration 900 / 1500: loss 2.302764
iteration 1000 / 1500: loss 2.302731
iteration 1100 / 1500: loss 2.302610
iteration 1200 / 1500: loss 2.302481
iteration 1300 / 1500: loss 2.302486
iteration 1400 / 1500: loss 2.302350
batch_size = 200, lr = 0.000010, hidden size = 100.000000, Valid_accuracy: 0.222000
```

```
iteration 0 / 1500: loss 2.304146
iteration 100 / 1500: loss 2.304027
iteration 200 / 1500: loss 2.304001
iteration 300 / 1500: loss 2.303894
iteration 400 / 1500: loss 2.303679
iteration 500 / 1500: loss 2.303553
iteration 600 / 1500: loss 2.303569
iteration 700 / 1500: loss 2.303473
iteration 800 / 1500: loss 2.303486
iteration 900 / 1500: loss 2.302919
iteration 1000 / 1500: loss 2.302718
iteration 1100 / 1500: loss 2.302805
iteration 1200 / 1500: loss 2.302672
iteration 1300 / 1500: loss 2.302825
iteration 1400 / 1500: loss 2.301850
batch_size = 200, lr = 0.000010, hidden size = 200.000000, Valid_accuracy: 0.208000
iteration 0 / 1500: loss 2.302956
iteration 100 / 1500: loss 1.908739
iteration 200 / 1500: loss 1.725849
iteration 300 / 1500: loss 1.747936
iteration 400 / 1500: loss 1.688771
iteration 500 / 1500: loss 1.602856
iteration 600 / 1500: loss 1.590805
iteration 700 / 1500: loss 1.570828
iteration 800 / 1500: loss 1.535409
iteration 900 / 1500: loss 1.573022
iteration 1000 / 1500: loss 1.536511
iteration 1100 / 1500: loss 1.441498
iteration 1200 / 1500: loss 1.508587
iteration 1300 / 1500: loss 1.338997
iteration 1400 / 1500: loss 1.502504
batch_size = 300, lr = 0.001000, hidden size = 50.000000, Valid_accuracy: 0.484000
iteration 0 / 1500: loss 2.303344
iteration 100 / 1500: loss 1.978512
iteration 200 / 1500: loss 1.749057
iteration 300 / 1500: loss 1.654061
iteration 400 / 1500: loss 1.634930
iteration 500 / 1500: loss 1.696440
iteration 600 / 1500: loss 1.648615
iteration 700 / 1500: loss 1.641480
iteration 800 / 1500: loss 1.457604
iteration 900 / 1500: loss 1.495703
iteration 1000 / 1500: loss 1.414465
iteration 1100 / 1500: loss 1.444840
iteration 1200 / 1500: loss 1.433139
iteration 1300 / 1500: loss 1.402709
iteration 1400 / 1500: loss 1.437508
batch_size = 300, lr = 0.001000, hidden size = 100.000000, Valid_accuracy: 0.496000
```

```
iteration 0 / 1500: loss 2.304183
iteration 100 / 1500: loss 1.906383
iteration 200 / 1500: loss 1.746232
iteration 300 / 1500: loss 1.777423
iteration 400 / 1500: loss 1.731506
iteration 500 / 1500: loss 1.541554
iteration 600 / 1500: loss 1.435085
iteration 700 / 1500: loss 1.577967
iteration 800 / 1500: loss 1.526153
iteration 900 / 1500: loss 1.537568
iteration 1000 / 1500: loss 1.514626
iteration 1100 / 1500: loss 1.485386
iteration 1200 / 1500: loss 1.447134
iteration 1300 / 1500: loss 1.307312
iteration 1400 / 1500: loss 1.378661
batch_size = 300, lr = 0.001000, hidden size = 200.000000, Valid_accuracy: 0.516000
iteration 0 / 1500: loss 2.303005
iteration 100 / 1500: loss 2.302665
iteration 200 / 1500: loss 2.299644
iteration 300 / 1500: loss 2.276634
iteration 400 / 1500: loss 2.192920
iteration 500 / 1500: loss 2.165777
iteration 600 / 1500: loss 2.100713
iteration 700 / 1500: loss 2.061189
iteration 800 / 1500: loss 2.061024
iteration 900 / 1500: loss 1.944629
iteration 1000 / 1500: loss 1.934184
iteration 1100 / 1500: loss 1.874635
iteration 1200 / 1500: loss 2.000557
iteration 1300 / 1500: loss 1.793956
iteration 1400 / 1500: loss 1.867173
batch_size = 300, lr = 0.000100, hidden size = 50.000000, Valid_accuracy: 0.340000
iteration 0 / 1500: loss 2.303339
iteration 100 / 1500: loss 2.302551
iteration 200 / 1500: loss 2.296450
iteration 300 / 1500: loss 2.253297
iteration 400 / 1500: loss 2.162080
iteration 500 / 1500: loss 2.099129
iteration 600 / 1500: loss 2.065859
iteration 700 / 1500: loss 2.108461
iteration 800 / 1500: loss 2.014786
iteration 900 / 1500: loss 1.951304
iteration 1000 / 1500: loss 1.896752
iteration 1100 / 1500: loss 1.934005
iteration 1200 / 1500: loss 1.886652
iteration 1300 / 1500: loss 1.829250
iteration 1400 / 1500: loss 1.792800
batch_size = 300, lr = 0.000100, hidden size = 100.000000, Valid_accuracy: 0.352000
```

```
iteration 0 / 1500: loss 2.304132
iteration 100 / 1500: loss 2.302399
iteration 200 / 1500: loss 2.292624
iteration 300 / 1500: loss 2.228037
iteration 400 / 1500: loss 2.144590
iteration 500 / 1500: loss 2.069490
iteration 600 / 1500: loss 2.035125
iteration 700 / 1500: loss 2.135214
iteration 800 / 1500: loss 1.945386
iteration 900 / 1500: loss 1.905479
iteration 1000 / 1500: loss 1.936170
iteration 1100 / 1500: loss 1.845281
iteration 1200 / 1500: loss 1.836648
iteration 1300 / 1500: loss 1.972798
iteration 1400 / 1500: loss 1.794804
batch_size = 300, lr = 0.000100, hidden size = 200.000000, Valid_accuracy: 0.356000
iteration 0 / 1500: loss 2.302973
iteration 100 / 1500: loss 2.302956
iteration 200 / 1500: loss 2.302936
iteration 300 / 1500: loss 2.302929
iteration 400 / 1500: loss 2.302900
iteration 500 / 1500: loss 2.302824
iteration 600 / 1500: loss 2.302803
iteration 700 / 1500: loss 2.302809
iteration 800 / 1500: loss 2.302694
iteration 900 / 1500: loss 2.302641
iteration 1000 / 1500: loss 2.302557
iteration 1100 / 1500: loss 2.302484
iteration 1200 / 1500: loss 2.302244
iteration 1300 / 1500: loss 2.302139
iteration 1400 / 1500: loss 2.301960
batch_size = 300, lr = 0.000010, hidden size = 50.000000, Valid_accuracy: 0.191000
iteration 0 / 1500: loss 2.303370
iteration 100 / 1500: loss 2.303292
iteration 200 / 1500: loss 2.303258
iteration 300 / 1500: loss 2.303180
iteration 400 / 1500: loss 2.303096
iteration 500 / 1500: loss 2.303103
iteration 600 / 1500: loss 2.302987
iteration 700 / 1500: loss 2.302824
iteration 800 / 1500: loss 2.302774
iteration 900 / 1500: loss 2.302552
iteration 1000 / 1500: loss 2.302378
iteration 1100 / 1500: loss 2.302263
iteration 1200 / 1500: loss 2.301925
iteration 1300 / 1500: loss 2.301772
iteration 1400 / 1500: loss 2.301180
batch_size = 300, lr = 0.000010, hidden size = 100.000000, Valid_accuracy: 0.188000
```

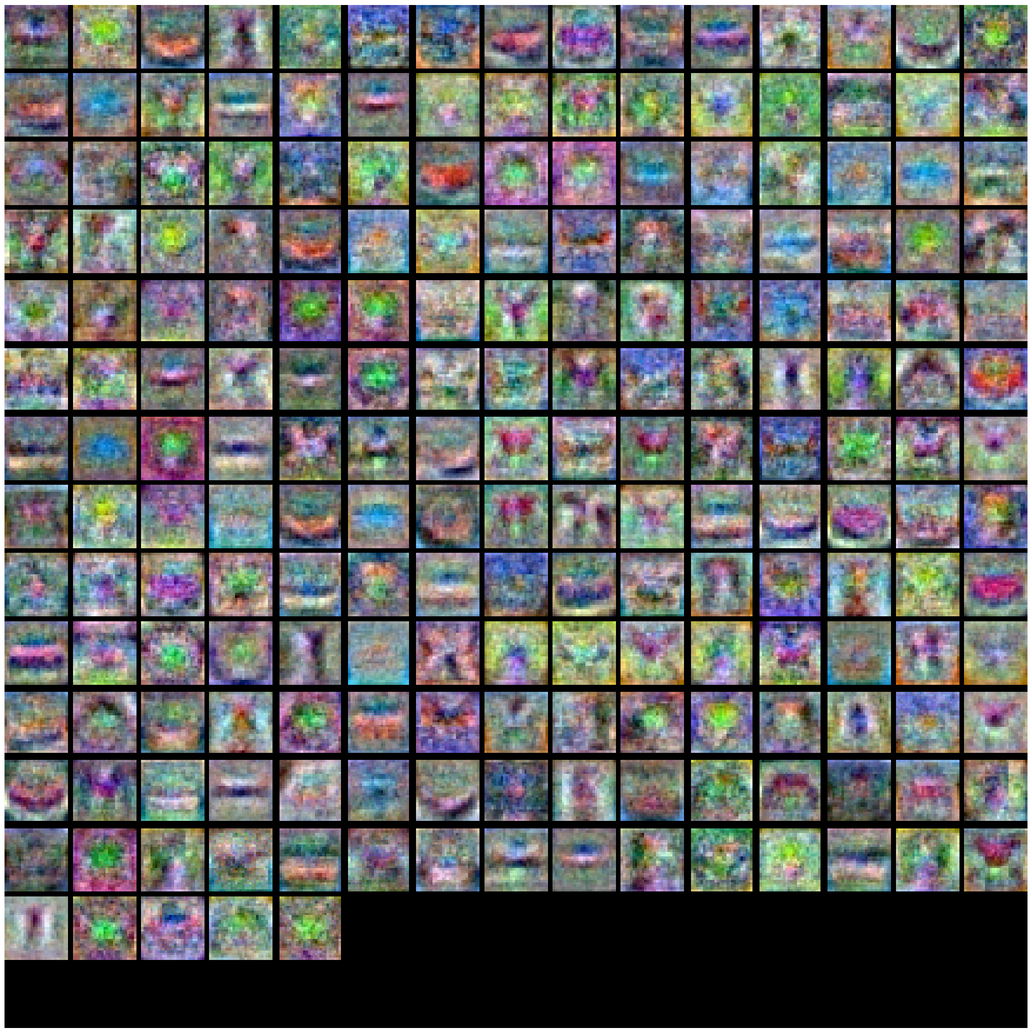
```
iteration 0 / 1500: loss 2.304084
iteration 100 / 1500: loss 2.304040
iteration 200 / 1500: loss 2.303916
iteration 300 / 1500: loss 2.303789
iteration 400 / 1500: loss 2.303728
iteration 500 / 1500: loss 2.303593
iteration 600 / 1500: loss 2.303439
iteration 700 / 1500: loss 2.303373
iteration 800 / 1500: loss 2.303034
iteration 900 / 1500: loss 2.302668
iteration 1000 / 1500: loss 2.302438
iteration 1100 / 1500: loss 2.301703
iteration 1200 / 1500: loss 2.301624
iteration 1300 / 1500: loss 2.301346
iteration 1400 / 1500: loss 2.301362
batch_size = 300, lr = 0.000010, hidden size = 200.000000, Valid_accuracy: 0.234000
iteration 0 / 1500: loss 2.302973
iteration 100 / 1500: loss 1.923041
iteration 200 / 1500: loss 1.807071
iteration 300 / 1500: loss 1.713001
iteration 400 / 1500: loss 1.632652
iteration 500 / 1500: loss 1.559257
iteration 600 / 1500: loss 1.503148
iteration 700 / 1500: loss 1.581622
iteration 800 / 1500: loss 1.544833
iteration 900 / 1500: loss 1.464013
iteration 1000 / 1500: loss 1.531469
iteration 1100 / 1500: loss 1.560648
iteration 1200 / 1500: loss 1.589543
iteration 1300 / 1500: loss 1.538699
iteration 1400 / 1500: loss 1.530522
batch_size = 400, lr = 0.001000, hidden size = 50.000000, Valid_accuracy: 0.485000
iteration 0 / 1500: loss 2.303355
iteration 100 / 1500: loss 1.987476
iteration 200 / 1500: loss 1.808412
iteration 300 / 1500: loss 1.739255
iteration 400 / 1500: loss 1.664695
iteration 500 / 1500: loss 1.557069
iteration 600 / 1500: loss 1.552847
iteration 700 / 1500: loss 1.468661
iteration 800 / 1500: loss 1.580804
iteration 900 / 1500: loss 1.538730
iteration 1000 / 1500: loss 1.487562
iteration 1100 / 1500: loss 1.496476
iteration 1200 / 1500: loss 1.506914
iteration 1300 / 1500: loss 1.482608
iteration 1400 / 1500: loss 1.528092
batch_size = 400, lr = 0.001000, hidden size = 100.000000, Valid_accuracy: 0.490000
```

```
iteration 0 / 1500: loss 2.304151
iteration 100 / 1500: loss 1.892961
iteration 200 / 1500: loss 1.751468
iteration 300 / 1500: loss 1.676551
iteration 400 / 1500: loss 1.635151
iteration 500 / 1500: loss 1.553707
iteration 600 / 1500: loss 1.592815
iteration 700 / 1500: loss 1.594222
iteration 800 / 1500: loss 1.485915
iteration 900 / 1500: loss 1.488332
iteration 1000 / 1500: loss 1.446905
iteration 1100 / 1500: loss 1.403172
iteration 1200 / 1500: loss 1.511301
iteration 1300 / 1500: loss 1.301931
iteration 1400 / 1500: loss 1.485221
batch_size = 400, lr = 0.001000, hidden size = 200.000000, Valid_accuracy: 0.512000
iteration 0 / 1500: loss 2.302972
iteration 100 / 1500: loss 2.302618
iteration 200 / 1500: loss 2.299602
iteration 300 / 1500: loss 2.268603
iteration 400 / 1500: loss 2.229501
iteration 500 / 1500: loss 2.165550
iteration 600 / 1500: loss 2.115693
iteration 700 / 1500: loss 2.042554
iteration 800 / 1500: loss 2.090412
iteration 900 / 1500: loss 2.005043
iteration 1000 / 1500: loss 2.002546
iteration 1100 / 1500: loss 1.945854
iteration 1200 / 1500: loss 1.926894
iteration 1300 / 1500: loss 1.888736
iteration 1400 / 1500: loss 1.923308
batch_size = 400, lr = 0.000100, hidden size = 50.000000, Valid_accuracy: 0.301000
iteration 0 / 1500: loss 2.303332
iteration 100 / 1500: loss 2.302510
iteration 200 / 1500: loss 2.295710
iteration 300 / 1500: loss 2.258166
iteration 400 / 1500: loss 2.181479
iteration 500 / 1500: loss 2.105668
iteration 600 / 1500: loss 2.086265
iteration 700 / 1500: loss 2.048544
iteration 800 / 1500: loss 1.977102
iteration 900 / 1500: loss 1.983210
iteration 1000 / 1500: loss 1.955787
iteration 1100 / 1500: loss 1.953908
iteration 1200 / 1500: loss 1.891762
iteration 1300 / 1500: loss 1.882815
iteration 1400 / 1500: loss 1.830816
batch_size = 400, lr = 0.000100, hidden size = 100.000000, Valid_accuracy: 0.319000
```

```
iteration 0 / 1500: loss 2.304117
iteration 100 / 1500: loss 2.302532
iteration 200 / 1500: loss 2.291602
iteration 300 / 1500: loss 2.211597
iteration 400 / 1500: loss 2.162270
iteration 500 / 1500: loss 2.126588
iteration 600 / 1500: loss 2.067392
iteration 700 / 1500: loss 2.043353
iteration 800 / 1500: loss 1.963780
iteration 900 / 1500: loss 1.971389
iteration 1000 / 1500: loss 1.937339
iteration 1100 / 1500: loss 1.932497
iteration 1200 / 1500: loss 1.911858
iteration 1300 / 1500: loss 1.959853
iteration 1400 / 1500: loss 1.860775
batch_size = 400, lr = 0.000100, hidden size = 200.000000, Valid_accuracy: 0.324000
iteration 0 / 1500: loss 2.302981
iteration 100 / 1500: loss 2.302951
iteration 200 / 1500: loss 2.302939
iteration 300 / 1500: loss 2.302922
iteration 400 / 1500: loss 2.302902
iteration 500 / 1500: loss 2.302895
iteration 600 / 1500: loss 2.302877
iteration 700 / 1500: loss 2.302857
iteration 800 / 1500: loss 2.302848
iteration 900 / 1500: loss 2.302807
iteration 1000 / 1500: loss 2.302783
iteration 1100 / 1500: loss 2.302762
iteration 1200 / 1500: loss 2.302711
iteration 1300 / 1500: loss 2.302719
iteration 1400 / 1500: loss 2.302656
batch_size = 400, lr = 0.000010, hidden size = 50.000000, Valid_accuracy: 0.238000
iteration 0 / 1500: loss 2.303350
iteration 100 / 1500: loss 2.303290
iteration 200 / 1500: loss 2.303252
iteration 300 / 1500: loss 2.303210
iteration 400 / 1500: loss 2.303154
iteration 500 / 1500: loss 2.303140
iteration 600 / 1500: loss 2.302996
iteration 700 / 1500: loss 2.302890
iteration 800 / 1500: loss 2.302894
iteration 900 / 1500: loss 2.302781
iteration 1000 / 1500: loss 2.302664
iteration 1100 / 1500: loss 2.302710
iteration 1200 / 1500: loss 2.302594
iteration 1300 / 1500: loss 2.302315
iteration 1400 / 1500: loss 2.302336
batch_size = 400, lr = 0.000010, hidden size = 100.000000, Valid_accuracy: 0.240000
```

```
iteration 0 / 1500: loss 2.304152
iteration 100 / 1500: loss 2.304044
iteration 200 / 1500: loss 2.303982
iteration 300 / 1500: loss 2.303911
iteration 400 / 1500: loss 2.303806
iteration 500 / 1500: loss 2.303609
iteration 600 / 1500: loss 2.303577
iteration 700 / 1500: loss 2.303386
iteration 800 / 1500: loss 2.303383
iteration 900 / 1500: loss 2.303267
iteration 1000 / 1500: loss 2.302850
iteration 1100 / 1500: loss 2.302886
iteration 1200 / 1500: loss 2.302542
iteration 1300 / 1500: loss 2.302545
iteration 1400 / 1500: loss 2.302022
batch_size = 400, lr = 0.000010, hidden size = 200.000000, Valid_accuracy: 0.230000
```

```
In [35]: # visualize the weights of the best network
         show_net_weights(best_net)
```



10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

```
In [36]: test_acc = (best_net.predict(X_test) == y_test).mean()
         print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.529
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply. 1. Train on a larger dataset. 2. Add more hidden units. 3. Increase the regularization strength. 4. None of the above.

Your answer: 1

Your explanation: We should expect the training accuracy much higher than the resting accuracy since the testing dataset is the dataset our model never see before, but the training dataset is the dataset the model used to train itself. More training data can help the model known better in the general aspects. But sometimes it doesn't help.