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 [161]:
```
 A bit of setup
rom __future__ import print_function

mport numpy as np
mport matplotlib.pyplot as plt

rom cs175.classifiers.neural_net import TwoLayerNet



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

 for auto-reloading external modules
 see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
on
load_ext autoreload
autoreload 2

ef 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 `cs175/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 [162]:
```
 Create a small net and some toy data to check your implementations.
 Note that we set the random seed for repeatable experiments.

nput_size = 4
idden_size = 10
um_classes = 3
um_inputs = 5

ef init_toy_model():
   np.random.seed(0)
   return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

ef 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

et = init_toy_model()
, y = init_toy_data()
```

Forward pass: compute scores

Open the file `cs175/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 [163]:
```
cores = net.loss(X)
rint('Your scores:')
rint(scores)
rint()
rint('correct scores:')
orrect_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]])
rint(correct_scores)
rint()

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

```
our 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]]

orrect 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]]

ifference between your scores and correct scores:
.68027209324e-08
```

Forward pass: compute loss

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

In [164]:
```
oss, _ = net.loss(X, y, reg=0.05)
orrect_loss = 1.30378789133

 should be very small, we get < 1e-12
rint('Difference between your loss and correct loss:')
rint(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.79412040779e-13
```

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 [165]:

```
rom cs175.gradient_check import eval_numerical_gradient

 Use numeric gradient checking to check your implementation of the backward p
ss.
 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.

oss, grads = net.loss(X, y, reg=0.05)

 these should all be less than 1e-8 or so
or 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], verbos
=False)
   print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
rads[param_name])))
```

```
b2 max relative error: 3.865070e-11
b1 max relative error: 1.555470e-09
W1 max relative error: 5.734357e-09
W2 max relative error: 3.440708e-09
```

Train the network

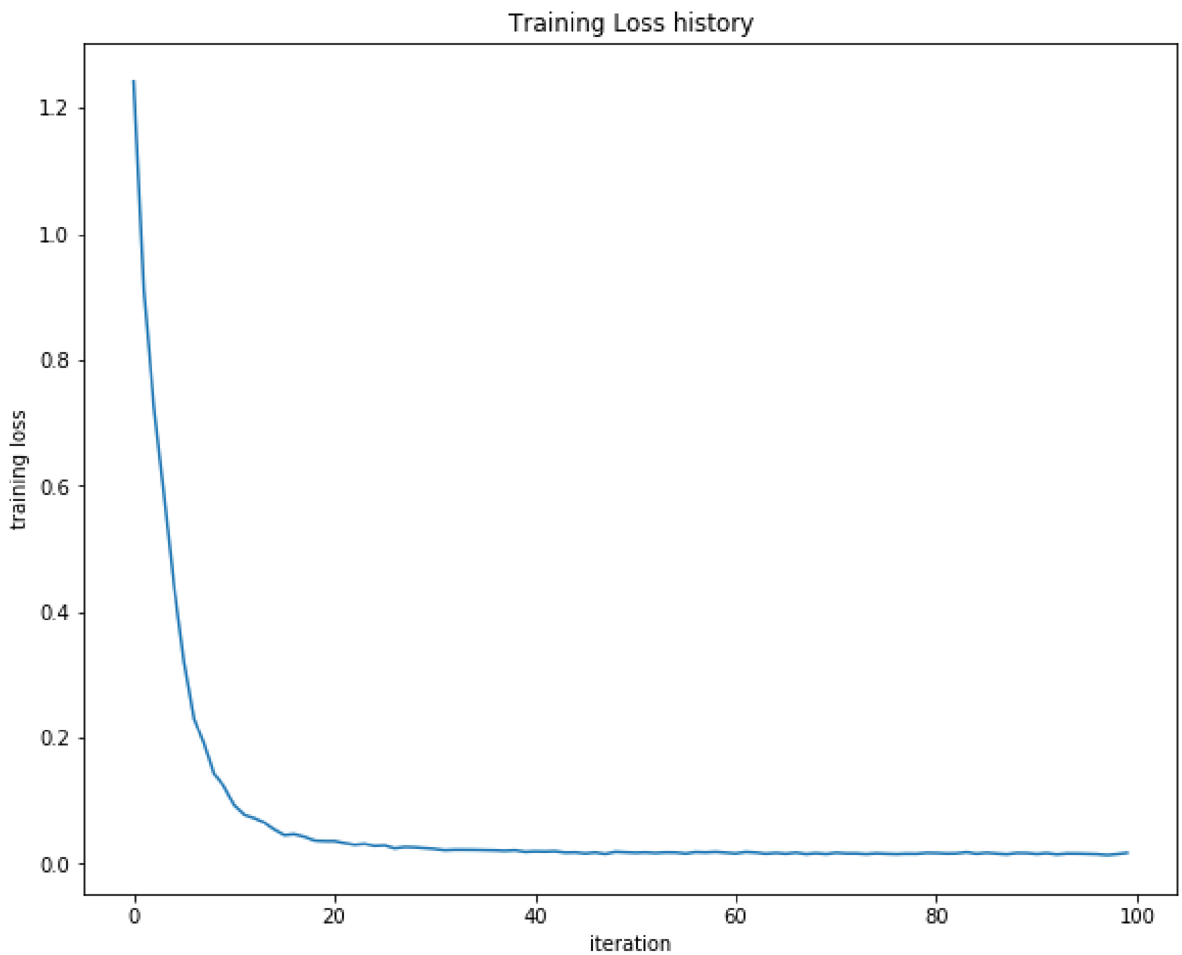
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and 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 [166]:

```
et = init_toy_model()
tats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.0171496079387
```



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 [167]:

```
rom cs175.data_utils import load_CIFAR10

ef 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 = 'CS175/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


 Invoke the above function to get our data.
_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
rint('Train data shape: ', X_train.shape)
rint('Train labels shape: ', y_train.shape)
rint('Validation data shape: ', X_val.shape)
rint('Validation labels shape: ', y_val.shape)
rint('Test data shape: ', X_test.shape)
rint('Test labels shape: ', y_test.shape)
```

```
rain data shape:  (49000L, 3072L)
rain labels shape:  (49000L,)
alidation data shape:  (1000L, 3072L)
alidation labels shape:  (1000L,)
est data shape:  (1000L, 3072L)
est labels shape:  (1000L,)
```

Train a network

To train our network we will use SGD with momentum. 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 [168]:
```
nput_size = 32 * 32 * 3
idden_size = 50
um_classes = 10
et = TwoLayerNet(input_size, hidden_size, num_classes)

 Train the network
tats = 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
al_acc = (net.predict(X_val) == y_val).mean()
rint('Validation accuracy: ', val_acc)
```

```
teration 0 / 1000: loss 2.302954
teration 100 / 1000: loss 2.302550
teration 200 / 1000: loss 2.297648
teration 300 / 1000: loss 2.259602
teration 400 / 1000: loss 2.204170
teration 500 / 1000: loss 2.118565
teration 600 / 1000: loss 2.051535
teration 700 / 1000: loss 1.988466
teration 800 / 1000: loss 2.006591
teration 900 / 1000: loss 1.951473
alidation accuracy:  0.287
```

Debug the training

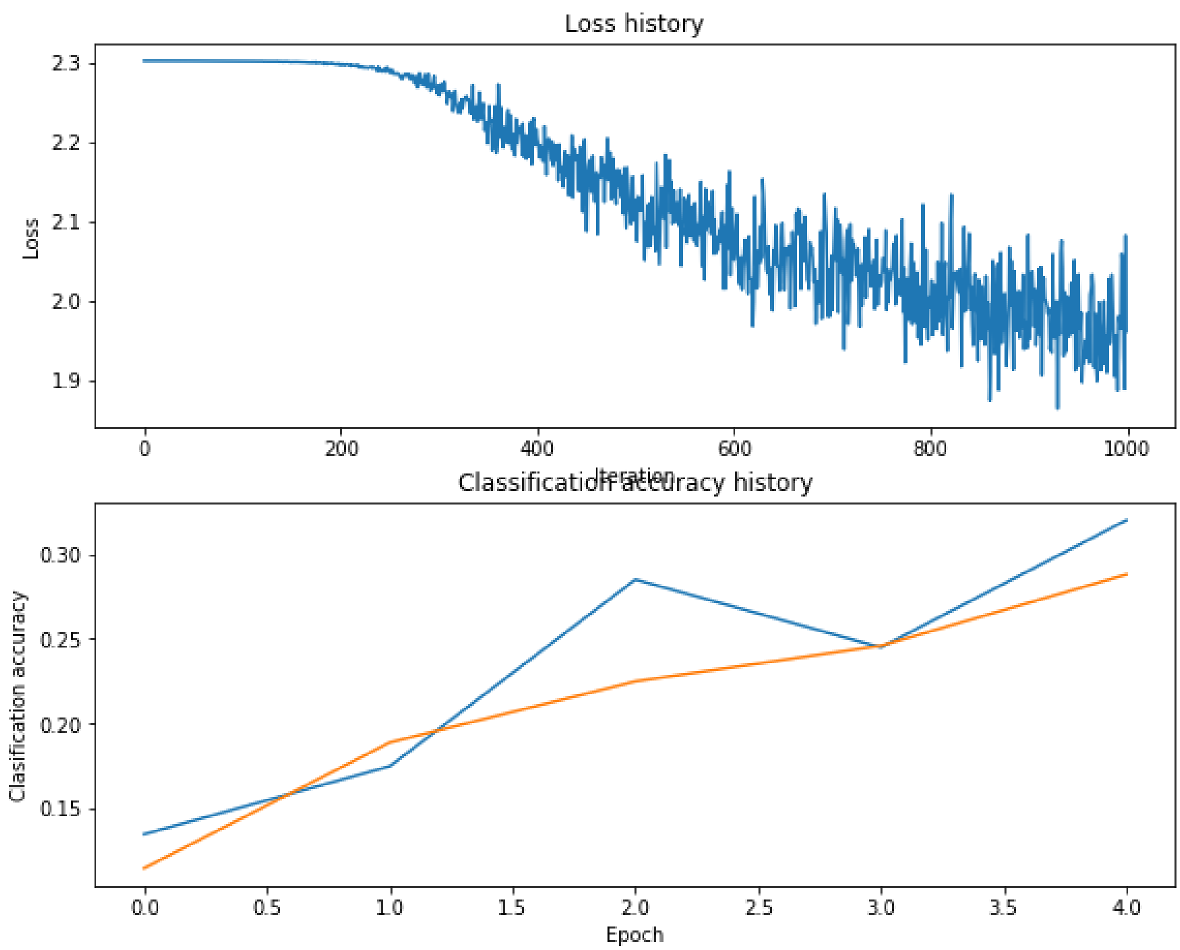
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

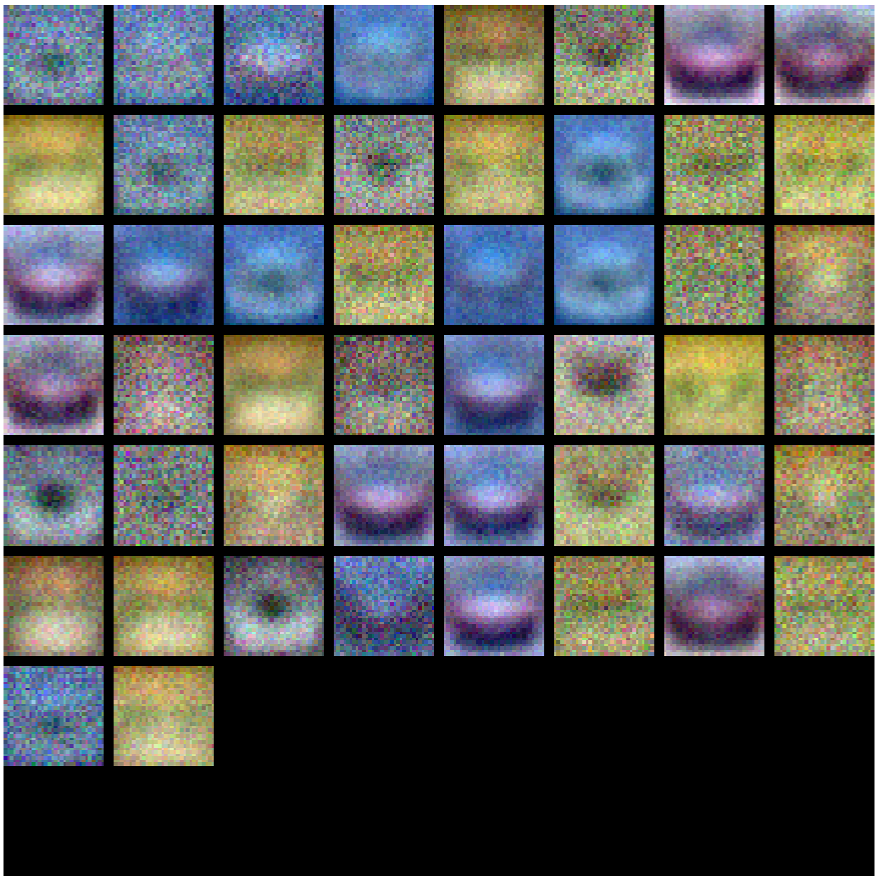
In [169]:

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

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



In [170]:

```
rom cs175.vis_utils import visualize_grid

 Visualize the weights of the network

ef 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()

how_net_weights(net)
```



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, number 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.

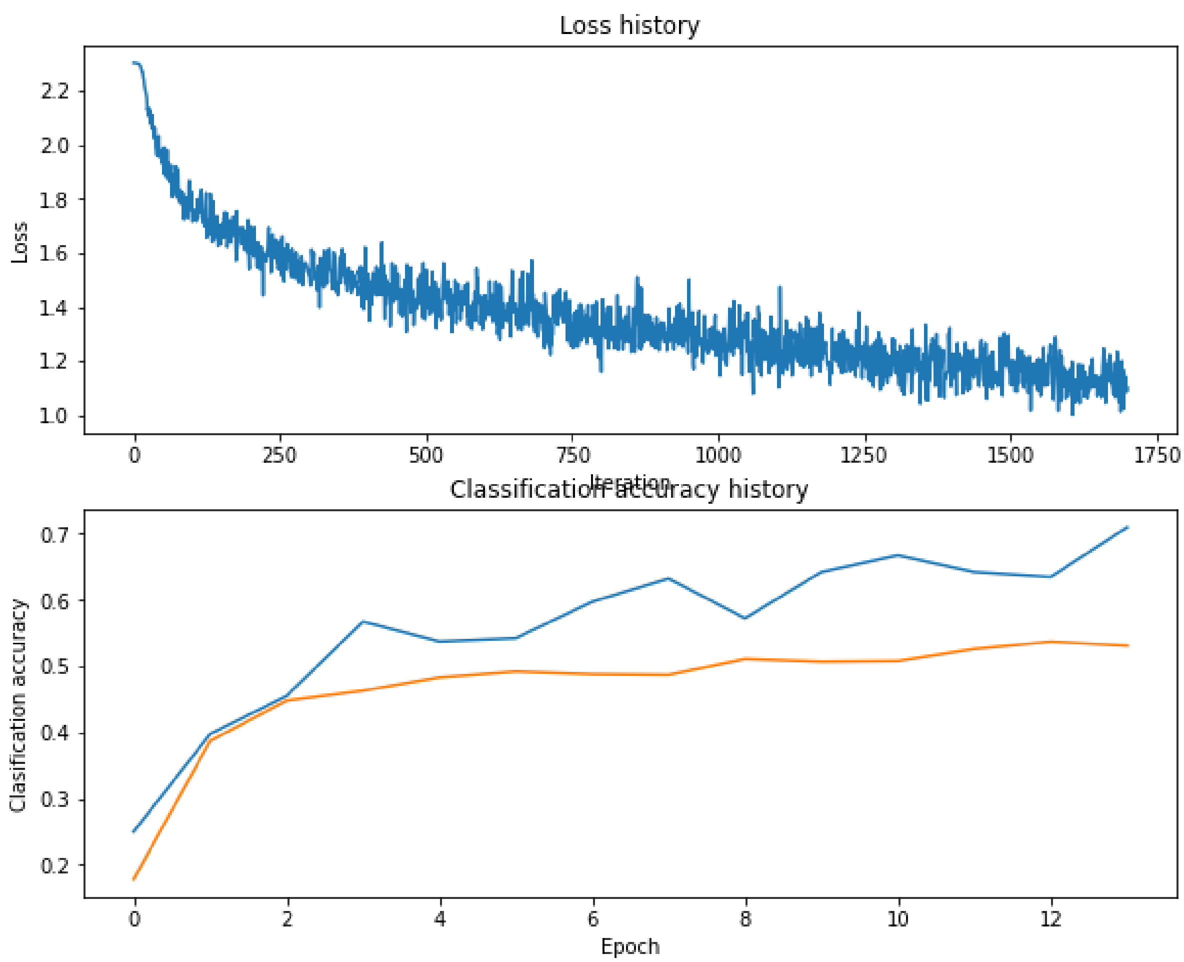
Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 52% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [171]:

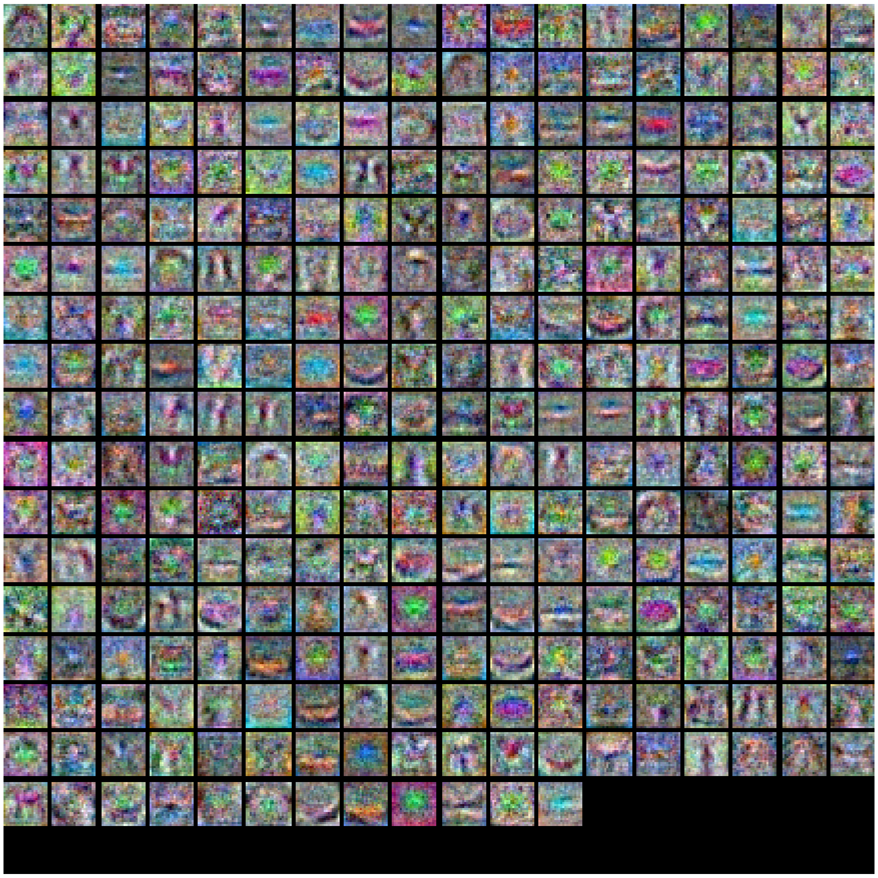
```
est_net = None # store the best model into this

################################################################################
##
 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.

################################################################################
##
atch_size = [400]
earning_rates = [1.6e-3]
egularization_strengths = [1e-2]
idden_size = 300
est_acc = -1
or bs in batch_size:
   print("Batch: ", bs)
   for learn in learning_rates:
       print("learn rate: ", learn)
       for reg in regularization_strengths:
           print("reg: ", reg)
           net = TwoLayerNet(input_size, hidden_size, num_classes)
           stats = net.train(X_train,y_train,X_val,y_val,num_iters=1700,batch
_size=bs,learning_rate=learn,reg=reg,verbose=True)
           val_acc = (net.predict(X_val) == y_val).mean()
           if val_acc > best_acc:
               best_net = net
               best_acc = val_acc

           # 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.show()
############################################################################
##
                               END OF YOUR CODE

############################################################################
##
```

```
Batch:  400
learn rate:  0.0016
reg:  0.01
iteration 0 / 1700: loss 2.302677
iteration 100 / 1700: loss 1.716000
iteration 200 / 1700: loss 1.689613
iteration 300 / 1700: loss 1.538780
iteration 400 / 1700: loss 1.428758
iteration 500 / 1700: loss 1.537459
iteration 600 / 1700: loss 1.350383
iteration 700 / 1700: loss 1.308230
iteration 800 / 1700: loss 1.160664
iteration 900 / 1700: loss 1.282771
iteration 1000 / 1700: loss 1.309236
iteration 1100 / 1700: loss 1.265470
iteration 1200 / 1700: loss 1.253023
iteration 1300 / 1700: loss 1.280708
iteration 1400 / 1700: loss 1.162491
iteration 1500 / 1700: loss 1.111606
iteration 1600 / 1700: loss 1.134132
```



In [172]:
```
 visualize the weights of the best network
how_net_weights(best_net)
```



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%.

We will give you extra bonus point for every 1% of accuracy above 52%.

In [173]:
```
est_acc = (best_net.predict(X_test) == y_test).mean()
rint('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.53
```