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 [20]:
```
# A bit of setup

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs682.classifiers.neural_net import TwoLayerNet


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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-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.ab
s(y))))
```

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

We will use the class `TwoLayerNet` in the file `cs682/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 [21]:
```
# 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()
```

Forward pass: compute scores

Open the file `cs682/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 [22]:
```
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:
3.6802720496109664e-08
```

Forward pass: compute loss

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

In [24]:
```
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:
1.794120407794253e-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 [25]:
```
from cs682.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the b
ackward pass.
# If your implementation is correct, the difference between the numer
ic 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[param_name])))
```

```
W1 max relative error: 3.561318e-09
W2 max relative error: 3.440708e-09
b1 max relative error: 2.738421e-09
b2 max relative error: 3.865070e-11
```

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.

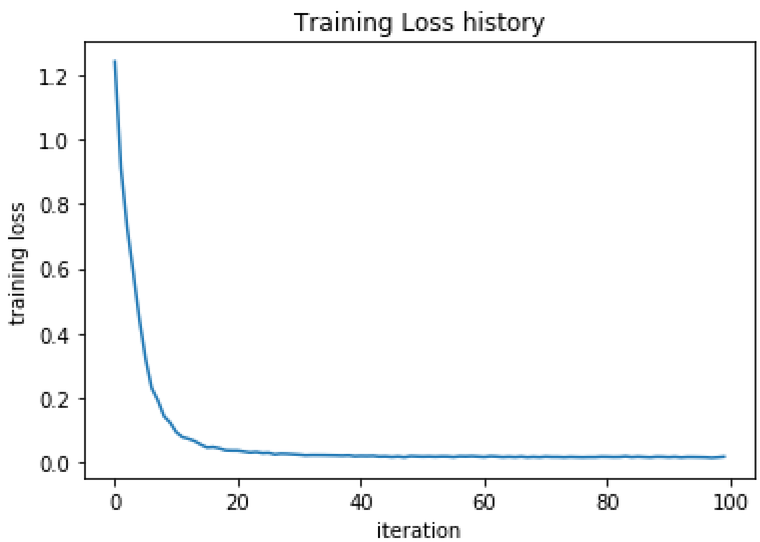
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

In [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:  0.017149607938732037
```



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 [9]:
```
from cs682.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_tes
t=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 st
eps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs682/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 memory issue)
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)
```

```
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,)
```

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 [26]:
```
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=3000, 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 / 3000: loss 2.302973
iteration 100 / 3000: loss 2.302542
iteration 200 / 3000: loss 2.298325
iteration 300 / 3000: loss 2.255561
iteration 400 / 3000: loss 2.182325
iteration 500 / 3000: loss 2.131531
iteration 600 / 3000: loss 2.126443
iteration 700 / 3000: loss 2.063506
iteration 800 / 3000: loss 2.040525
iteration 900 / 3000: loss 1.990341
iteration 1000 / 3000: loss 1.930121
iteration 1100 / 3000: loss 1.996365
iteration 1200 / 3000: loss 1.863751
iteration 1300 / 3000: loss 1.865116
iteration 1400 / 3000: loss 1.905642
iteration 1500 / 3000: loss 1.890493
iteration 1600 / 3000: loss 1.854396
iteration 1700 / 3000: loss 1.773150
iteration 1800 / 3000: loss 1.841489
iteration 1900 / 3000: loss 1.918407
iteration 2000 / 3000: loss 1.811878
iteration 2100 / 3000: loss 1.711787
iteration 2200 / 3000: loss 1.800602
iteration 2300 / 3000: loss 1.717463
iteration 2400 / 3000: loss 1.792638
iteration 2500 / 3000: loss 1.756333
iteration 2600 / 3000: loss 1.687884
iteration 2700 / 3000: loss 1.821500
iteration 2800 / 3000: loss 1.758978
iteration 2900 / 3000: loss 1.731975
Validation accuracy:  0.398
```

Debug the training

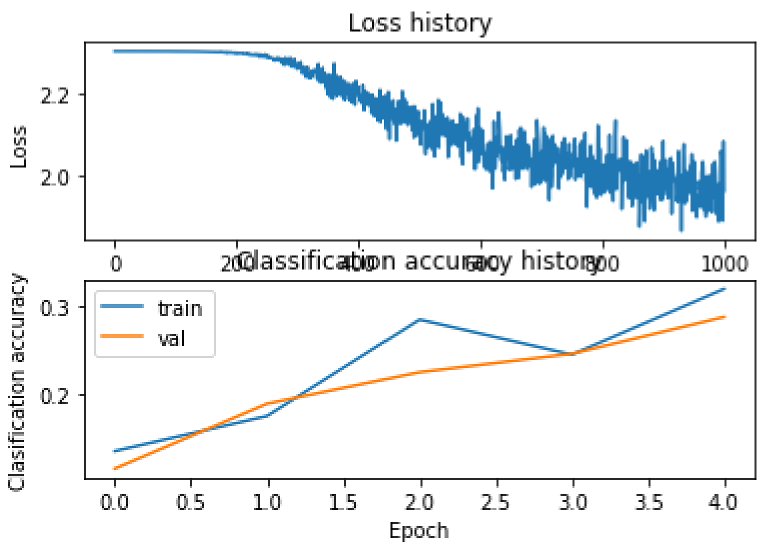
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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 [11]:
```
# 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()
```

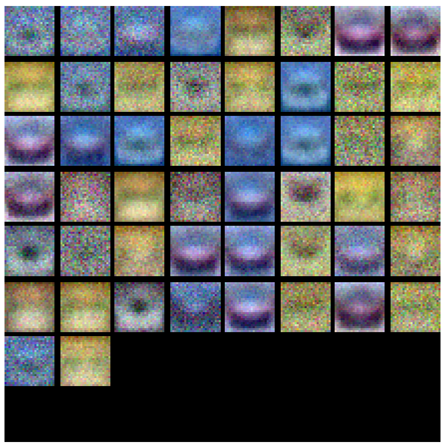


In [12]:
```
from cs682.vis_utils import visualize_grid

# 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)
```



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 [15]:
```
best_net = None # store the best model into this
input_size = 32 * 32 * 3
learning_rates = [1e-3, 3e-3]
regularization_strengths = [0.2, 0.5]
hiddenSizes = range(100,101)
num_classes = 10
results = {}
best_val = -1
best_lr = -1
best_reg = -1
best_hidden_size = -1
inner_iterations = 3000
batch_size = 200
learning_rate_decay=0.95

#####################################################################
############
# TODO: Tune hyperparameters using the validation set. Store your bes
t trained  #
# model in best_net.
#
#
#
# To help debug your network, it may help to use visualizations simil
ar to the  #
# ones we used above; these visualizations will have significant qual
itative    #
# differences from the ones we saw above for the poorly tuned networ
k.         #
#
#
# Tweaking hyperparameters by hand can be fun, but you might find it
 useful to  #
# write code to sweep through possible combinations of hyperparameter
s          #
# automatically like we did on the previous exercises.
#
#####################################################################
############
num_iters = 10
for it in range(num_iters):
    for jt in range(num_iters):
        for kt in range(len(hiddenSizes)):
            learning_rate = learning_rates[0] + it * ((learning_rates
[1] - learning_rates[0]) / num_iters)
            reg = regularization_strengths[0] + jt * ((regularization
_strengths[1] - regularization_strengths[0])/ num_iters)
            hidden_size = hiddenSizes[kt]

            net = TwoLayerNet(input_size, hidden_size, num_classes)
            # Train the network
            stats = net.train(X_train, y_train, X_val, y_val,
                        learning_rate, learning_rate_decay,
                        reg, inner_iterations, batch_size,verbose=Fal
se)
```

```
            y_train_pred = net.predict(X_train)
            y_val_pred = net.predict(X_val)
            training_accuracy = np.mean(y_train == y_train_pred)
            validation_accuracy = np.mean(y_val == y_val_pred)
            results[(learning_rate, reg, hidden_size)] = (training_ac
curacy, validation_accuracy)
            print('lr %e reg %e hidden %f train accuracy: %f val accu
racy: %f' % (
                        learning_rate, reg, hidden_size, training_acc
uracy, validation_accuracy))
            if validation_accuracy > best_val:
                best_val = validation_accuracy
                best_net = net
                best_lr = learning_rate
                best_reg = reg
                best_hidden_size = hidden_size

print('best validation accuracy achieved during cross-validation: lr
%e reg %e hidden %f val accuracy: %f, ' % (learning_rate, reg, hidden
_size, best_val))
######################################################################
############
#                               END OF YOUR CODE
#
######################################################################
############
```

```
lr 1.000000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
580367 val accuracy: 0.511000
lr 1.000000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
580776 val accuracy: 0.502000
lr 1.000000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
577776 val accuracy: 0.522000
lr 1.000000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
575959 val accuracy: 0.532000
lr 1.000000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
565367 val accuracy: 0.506000
lr 1.000000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
564122 val accuracy: 0.527000
lr 1.000000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
565306 val accuracy: 0.511000
lr 1.000000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
559551 val accuracy: 0.506000
lr 1.000000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
556673 val accuracy: 0.507000
lr 1.000000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
554041 val accuracy: 0.515000
lr 1.200000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
588694 val accuracy: 0.500000
lr 1.200000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
579306 val accuracy: 0.520000
lr 1.200000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
574980 val accuracy: 0.519000
lr 1.200000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
574633 val accuracy: 0.527000
lr 1.200000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
573122 val accuracy: 0.529000
lr 1.200000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
566327 val accuracy: 0.514000
lr 1.200000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
564653 val accuracy: 0.508000
lr 1.200000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
554449 val accuracy: 0.494000
lr 1.200000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
556388 val accuracy: 0.497000
lr 1.200000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
559306 val accuracy: 0.505000
lr 1.400000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
589898 val accuracy: 0.519000
lr 1.400000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
579367 val accuracy: 0.518000
lr 1.400000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
576204 val accuracy: 0.522000
lr 1.400000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
579429 val accuracy: 0.520000
lr 1.400000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
566388 val accuracy: 0.523000
lr 1.400000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
565694 val accuracy: 0.525000
lr 1.400000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
562306 val accuracy: 0.515000
lr 1.400000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
561918 val accuracy: 0.520000
lr 1.400000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
```

```
557939 val accuracy: 0.491000
lr 1.400000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
558286 val accuracy: 0.508000
lr 1.600000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
583571 val accuracy: 0.492000
lr 1.600000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
581898 val accuracy: 0.504000
lr 1.600000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
567286 val accuracy: 0.512000
lr 1.600000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
582857 val accuracy: 0.525000
lr 1.600000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
574816 val accuracy: 0.524000
lr 1.600000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
567429 val accuracy: 0.516000
lr 1.600000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
552041 val accuracy: 0.500000
lr 1.600000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
551143 val accuracy: 0.511000
lr 1.600000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
544633 val accuracy: 0.496000
lr 1.600000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
541673 val accuracy: 0.491000
lr 1.800000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
577694 val accuracy: 0.509000
lr 1.800000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
582490 val accuracy: 0.530000
lr 1.800000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
540673 val accuracy: 0.493000
lr 1.800000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
568980 val accuracy: 0.519000
lr 1.800000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
559653 val accuracy: 0.512000
lr 1.800000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
552347 val accuracy: 0.495000
lr 1.800000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
557245 val accuracy: 0.495000
lr 1.800000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
562939 val accuracy: 0.521000
lr 1.800000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
551286 val accuracy: 0.506000
lr 1.800000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
551878 val accuracy: 0.509000
lr 2.000000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
592306 val accuracy: 0.517000
lr 2.000000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
574776 val accuracy: 0.499000
lr 2.000000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
560857 val accuracy: 0.502000
lr 2.000000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
560694 val accuracy: 0.492000
lr 2.000000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
569959 val accuracy: 0.508000
lr 2.000000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
559408 val accuracy: 0.504000
lr 2.000000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
556245 val accuracy: 0.503000
```

```
lr 2.000000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
558673 val accuracy: 0.521000
lr 2.000000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
553184 val accuracy: 0.507000
lr 2.000000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
547592 val accuracy: 0.509000
lr 2.200000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
579143 val accuracy: 0.505000
lr 2.200000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
572122 val accuracy: 0.530000
lr 2.200000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
582816 val accuracy: 0.529000
lr 2.200000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
551776 val accuracy: 0.509000
lr 2.200000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
561694 val accuracy: 0.498000
lr 2.200000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
556612 val accuracy: 0.503000
lr 2.200000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
554408 val accuracy: 0.511000
lr 2.200000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
541571 val accuracy: 0.504000
lr 2.200000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
546673 val accuracy: 0.497000
lr 2.200000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
545633 val accuracy: 0.495000
lr 2.400000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
584041 val accuracy: 0.516000
lr 2.400000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
570592 val accuracy: 0.495000
lr 2.400000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
568612 val accuracy: 0.514000
lr 2.400000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
568020 val accuracy: 0.503000
lr 2.400000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
558714 val accuracy: 0.496000
lr 2.400000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
544776 val accuracy: 0.487000
lr 2.400000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
536061 val accuracy: 0.494000
lr 2.400000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
544571 val accuracy: 0.500000
lr 2.400000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
541776 val accuracy: 0.502000
lr 2.400000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
532510 val accuracy: 0.496000
lr 2.600000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
568633 val accuracy: 0.505000
lr 2.600000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
556959 val accuracy: 0.505000
lr 2.600000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
551429 val accuracy: 0.504000
lr 2.600000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
525796 val accuracy: 0.484000
lr 2.600000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
544041 val accuracy: 0.502000
lr 2.600000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
```

```
541735 val accuracy: 0.494000
lr 2.600000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
549735 val accuracy: 0.495000
lr 2.600000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
539551 val accuracy: 0.513000
lr 2.600000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
531857 val accuracy: 0.492000
lr 2.600000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
540857 val accuracy: 0.516000
lr 2.800000e-03 reg 2.000000e-01 hidden 100.000000 train accuracy: 0.
545755 val accuracy: 0.503000
lr 2.800000e-03 reg 2.300000e-01 hidden 100.000000 train accuracy: 0.
560449 val accuracy: 0.508000
lr 2.800000e-03 reg 2.600000e-01 hidden 100.000000 train accuracy: 0.
556102 val accuracy: 0.502000
lr 2.800000e-03 reg 2.900000e-01 hidden 100.000000 train accuracy: 0.
554327 val accuracy: 0.507000
lr 2.800000e-03 reg 3.200000e-01 hidden 100.000000 train accuracy: 0.
551388 val accuracy: 0.489000
lr 2.800000e-03 reg 3.500000e-01 hidden 100.000000 train accuracy: 0.
503163 val accuracy: 0.457000
lr 2.800000e-03 reg 3.800000e-01 hidden 100.000000 train accuracy: 0.
535837 val accuracy: 0.497000
lr 2.800000e-03 reg 4.100000e-01 hidden 100.000000 train accuracy: 0.
526571 val accuracy: 0.478000
lr 2.800000e-03 reg 4.400000e-01 hidden 100.000000 train accuracy: 0.
540592 val accuracy: 0.503000
lr 2.800000e-03 reg 4.700000e-01 hidden 100.000000 train accuracy: 0.
524082 val accuracy: 0.478000
best validation accuracy achieved during cross-validation: lr 2.80000
0e-03 reg 4.700000e-01 hidden 100.000000 val accuracy: 0.532000,
```

In [16]:
```
# visualize the weights of the best network
show_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%.

In [17]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.519
```

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: 1,3

Your explanation:

1. This is TRUE because Training on a large dataset might make the model more generalized and might decrease the gap between test and trainign data accuracy.
2. This is FALSE because adding more hidden layers might overfit the training data more.
3. This is TRUE because increasing the regularization strength might handle the overfitting problem more effectively.