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 [ ]:

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%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-ipytho
n
%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))))
```

We will use the class `TwoLayerNet` in the file `cs231n/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 [ ]:

```
# 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 `cs231n/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 [ ]:

```
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:
3.6802720745909845e-08
```

Forward pass: compute loss

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

In [ ]:

```
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:
1.7985612998927536e-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 [ ]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pas
s.
# 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, g
rads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447656e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-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 0.02.

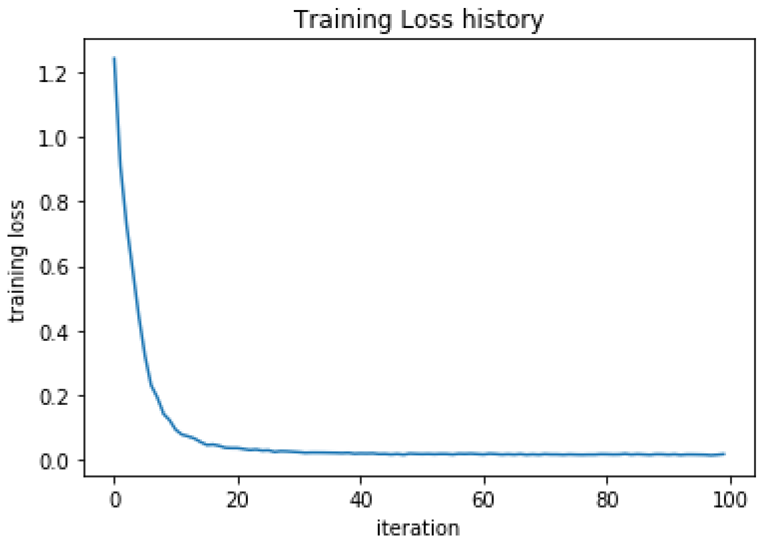
In [ ]:

```
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:  0.017149607938732083
```



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 [ ]:

```
from cs231n.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 = 'cs231n/datasets/cifar-10-batches-py'

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

    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,)
```

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 [ ]:

```
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 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation 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.

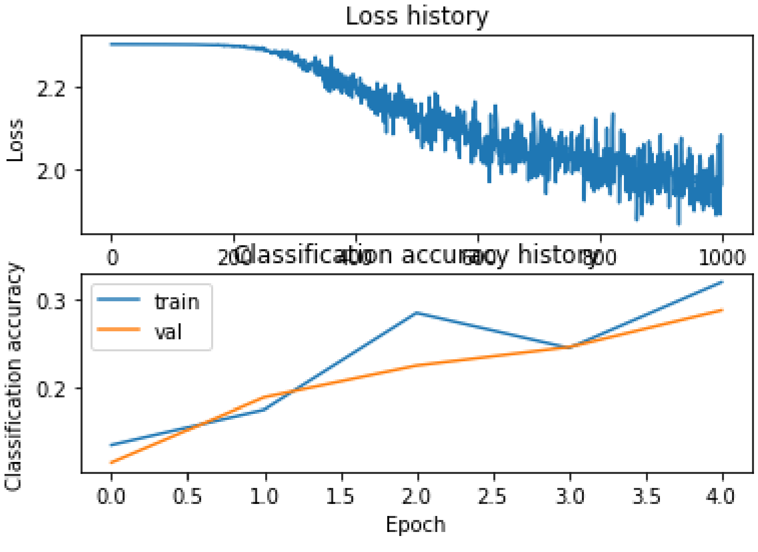
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [ ]:

```
# 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('Classification accuracy')
plt.legend()
plt.show()
```



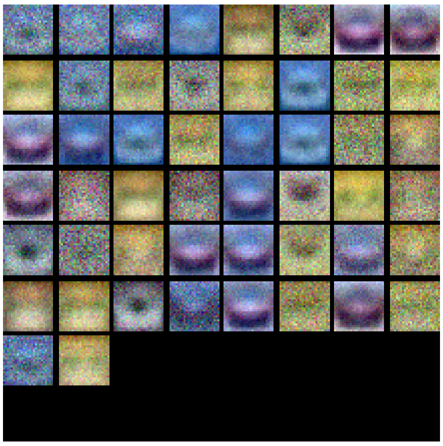
In [ ]:

```
from cs231n.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 (52% could serve as a reference), 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.).

Explain your hyperparameter tuning process below.

$\color{blue}{Your Answer:}$ list a combination of hyperparameter, use them to train the network and test on the validation set. Choose the hyperparameters with the best validation accuracy

In [ ]:

```
best_net = None # store the best model into this
best_accuracy = 0.0

################################################################################
#
# 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.
 #
################################################################################
#
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [75, 100,300, 500]
regs = [0.001, 0.01, 0.5, 1, 3]
lrs = [1e-4, 4e-4, 9e-4, 2e-3]


for hidden_size in hidden_sizes:
    for reg in regs:
        for lr in lrs:
            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=300,
                        learning_rate=lr, learning_rate_decay=0.95,
                        reg=reg, verbose=False)

            # Predict on the validation set
            train_acc = (net.predict(X_train) == y_train).mean()
            val_acc = (net.predict(X_val) == y_val).mean()

            print("hidden_size:", hidden_size, "reg:", reg, "lr:", lr, "val_ac
c:", val_acc, "train_acc:", train_acc)
            if (val_acc > best_accuracy):
                best_net = net
                best_accuracy = val_acc

print("best accuracy:", best_accuracy)
```

```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
hidden_size: 75 reg: 0.001 lr: 0.0001 val_acc: 0.306 train_acc: 0.30
159183673469386
hidden_size: 75 reg: 0.001 lr: 0.0004 val_acc: 0.455 train_acc: 0.44
918367346938776
hidden_size: 75 reg: 0.001 lr: 0.0009 val_acc: 0.477 train_acc: 0.50
01836734693877
hidden_size: 75 reg: 0.001 lr: 0.002 val_acc: 0.487 train_acc: 0.505
0816326530613
hidden_size: 75 reg: 0.01 lr: 0.0001 val_acc: 0.303 train_acc: 0.307
6938775510204
hidden_size: 75 reg: 0.01 lr: 0.0004 val_acc: 0.454 train_acc: 0.444
18367346938775
hidden_size: 75 reg: 0.01 lr: 0.0009 val_acc: 0.495 train_acc: 0.505
5714285714286
hidden_size: 75 reg: 0.01 lr: 0.002 val_acc: 0.457 train_acc: 0.5046
530612244898
hidden_size: 75 reg: 0.5 lr: 0.0001 val_acc: 0.306 train_acc: 0.3042
0408163265306
hidden_size: 75 reg: 0.5 lr: 0.0004 val_acc: 0.446 train_acc: 0.4388
1632653061226
hidden_size: 75 reg: 0.5 lr: 0.0009 val_acc: 0.462 train_acc: 0.4911
632653061225
hidden_size: 75 reg: 0.5 lr: 0.002 val_acc: 0.453 train_acc: 0.47155
102040816327
hidden_size: 75 reg: 1 lr: 0.0001 val_acc: 0.296 train_acc: 0.299693
8775510204
hidden_size: 75 reg: 1 lr: 0.0004 val_acc: 0.426 train_acc: 0.432081
6326530612
hidden_size: 75 reg: 1 lr: 0.0009 val_acc: 0.483 train_acc: 0.479795
9183673469
hidden_size: 75 reg: 1 lr: 0.002 val_acc: 0.469 train_acc: 0.4798571
4285714287
hidden_size: 75 reg: 3 lr: 0.0001 val_acc: 0.279 train_acc: 0.281857
14285714286
hidden_size: 75 reg: 3 lr: 0.0004 val_acc: 0.419 train_acc: 0.398979
5918367347
hidden_size: 75 reg: 3 lr: 0.0009 val_acc: 0.442 train_acc: 0.425122
44897959184
hidden_size: 75 reg: 3 lr: 0.002 val_acc: 0.438 train_acc: 0.4170408
163265306
hidden_size: 100 reg: 0.001 lr: 0.0001 val_acc: 0.306 train_acc: 0.3
0873469387755104
hidden_size: 100 reg: 0.001 lr: 0.0004 val_acc: 0.452 train_acc: 0.4
5010204081632654
hidden_size: 100 reg: 0.001 lr: 0.0009 val_acc: 0.47 train_acc: 0.51
75918367346939
hidden_size: 100 reg: 0.001 lr: 0.002 val_acc: 0.488 train_acc: 0.52
16938775510204
hidden_size: 100 reg: 0.01 lr: 0.0001 val_acc: 0.303 train_acc: 0.30
60816326530612
hidden_size: 100 reg: 0.01 lr: 0.0004 val_acc: 0.453 train_acc: 0.45
026530612244897
hidden_size: 100 reg: 0.01 lr: 0.0009 val_acc: 0.471 train_acc: 0.50
43877551020408
hidden_size: 100 reg: 0.01 lr: 0.002 val_acc: 0.48 train_acc: 0.5349
183673469388
hidden_size: 100 reg: 0.5 lr: 0.0001 val_acc: 0.305 train_acc: 0.307
83673469387757
hidden_size: 100 reg: 0.5 lr: 0.0004 val_acc: 0.451 train_acc: 0.444
42857142857145
hidden_size: 100 reg: 0.5 lr: 0.0009 val_acc: 0.48 train_acc: 0.4942
```

```
244897959184
hidden_size: 100 reg: 0.5 lr: 0.002 val_acc: 0.488 train_acc: 0.5037
755102040816
hidden_size: 100 reg: 1 lr: 0.0001 val_acc: 0.309 train_acc: 0.30444
897959183675
hidden_size: 100 reg: 1 lr: 0.0004 val_acc: 0.452 train_acc: 0.43716
326530612243
hidden_size: 100 reg: 1 lr: 0.0009 val_acc: 0.479 train_acc: 0.47722
448979591836
hidden_size: 100 reg: 1 lr: 0.002 val_acc: 0.47 train_acc: 0.4824693
8775510206
hidden_size: 100 reg: 3 lr: 0.0001 val_acc: 0.283 train_acc: 0.28940
81632653061
hidden_size: 100 reg: 3 lr: 0.0004 val_acc: 0.41 train_acc: 0.400612
24489795916
hidden_size: 100 reg: 3 lr: 0.0009 val_acc: 0.429 train_acc: 0.42806
122448979594
hidden_size: 100 reg: 3 lr: 0.002 val_acc: 0.427 train_acc: 0.414367
3469387755
hidden_size: 300 reg: 0.001 lr: 0.0001 val_acc: 0.316 train_acc: 0.3
2555102040816325
hidden_size: 300 reg: 0.001 lr: 0.0004 val_acc: 0.457 train_acc: 0.4
586938775510204
hidden_size: 300 reg: 0.001 lr: 0.0009 val_acc: 0.495 train_acc: 0.5
169795918367347
hidden_size: 300 reg: 0.001 lr: 0.002 val_acc: 0.422 train_acc: 0.45
63265306122449
hidden_size: 300 reg: 0.01 lr: 0.0001 val_acc: 0.313 train_acc: 0.32
49183673469388
hidden_size: 300 reg: 0.01 lr: 0.0004 val_acc: 0.462 train_acc: 0.45
74285714285714
hidden_size: 300 reg: 0.01 lr: 0.0009 val_acc: 0.487 train_acc: 0.52
71632653061225
hidden_size: 300 reg: 0.01 lr: 0.002 val_acc: 0.499 train_acc: 0.534
6122448979592
hidden_size: 300 reg: 0.5 lr: 0.0001 val_acc: 0.315 train_acc: 0.321
3061224489796
hidden_size: 300 reg: 0.5 lr: 0.0004 val_acc: 0.466 train_acc: 0.451
3469387755102
hidden_size: 300 reg: 0.5 lr: 0.0009 val_acc: 0.489 train_acc: 0.505
7755102040816
hidden_size: 300 reg: 0.5 lr: 0.002 val_acc: 0.484 train_acc: 0.5071
428571428571
hidden_size: 300 reg: 1 lr: 0.0001 val_acc: 0.315 train_acc: 0.31763
26530612245
hidden_size: 300 reg: 1 lr: 0.0004 val_acc: 0.449 train_acc: 0.43791
83673469388
hidden_size: 300 reg: 1 lr: 0.0009 val_acc: 0.469 train_acc: 0.47557
14285714286
hidden_size: 300 reg: 1 lr: 0.002 val_acc: 0.458 train_acc: 0.474836
73469387755
hidden_size: 300 reg: 3 lr: 0.0001 val_acc: 0.312 train_acc: 0.30646
9387755102
hidden_size: 300 reg: 3 lr: 0.0004 val_acc: 0.416 train_acc: 0.41108
163265306125
hidden_size: 300 reg: 3 lr: 0.0009 val_acc: 0.444 train_acc: 0.43075
510204081635
hidden_size: 300 reg: 3 lr: 0.002 val_acc: 0.425 train_acc: 0.423387
7551020408
hidden_size: 500 reg: 0.001 lr: 0.0001 val_acc: 0.323 train_acc: 0.3
296938775510204
```

```
hidden_size: 500 reg: 0.001 lr: 0.0004 val_acc: 0.465 train_acc: 0.4
6285714285714286
hidden_size: 500 reg: 0.001 lr: 0.0009 val_acc: 0.5 train_acc: 0.529
6326530612245
hidden_size: 500 reg: 0.001 lr: 0.002 val_acc: 0.501 train_acc: 0.55
92244897959183
hidden_size: 500 reg: 0.01 lr: 0.0001 val_acc: 0.319 train_acc: 0.32
95918367346939
hidden_size: 500 reg: 0.01 lr: 0.0004 val_acc: 0.467 train_acc: 0.46
344897959183673
hidden_size: 500 reg: 0.01 lr: 0.0009 val_acc: 0.492 train_acc: 0.53
2734693877551
hidden_size: 500 reg: 0.01 lr: 0.002 val_acc: 0.487 train_acc: 0.544
2040816326531
hidden_size: 500 reg: 0.5 lr: 0.0001 val_acc: 0.323 train_acc: 0.328
7551020408163
hidden_size: 500 reg: 0.5 lr: 0.0004 val_acc: 0.453 train_acc: 0.450
5918367346939
hidden_size: 500 reg: 0.5 lr: 0.0009 val_acc: 0.479 train_acc: 0.513
0408163265306
hidden_size: 500 reg: 0.5 lr: 0.002 val_acc: 0.498 train_acc: 0.5222
653061224489
hidden_size: 500 reg: 1 lr: 0.0001 val_acc: 0.319 train_acc: 0.32536
73469387755
hidden_size: 500 reg: 1 lr: 0.0004 val_acc: 0.452 train_acc: 0.44428
57142857143
hidden_size: 500 reg: 1 lr: 0.0009 val_acc: 0.461 train_acc: 0.48836
734693877554
hidden_size: 500 reg: 1 lr: 0.002 val_acc: 0.445 train_acc: 0.457
hidden_size: 500 reg: 3 lr: 0.0001 val_acc: 0.317 train_acc: 0.31381
632653061226
hidden_size: 500 reg: 3 lr: 0.0004 val_acc: 0.426 train_acc: 0.41197
95918367347
hidden_size: 500 reg: 3 lr: 0.0009 val_acc: 0.442 train_acc: 0.43861
22448979592
hidden_size: 500 reg: 3 lr: 0.002 val_acc: 0.442 train_acc: 0.425122
44897959184
best accuracy: 0.501
```

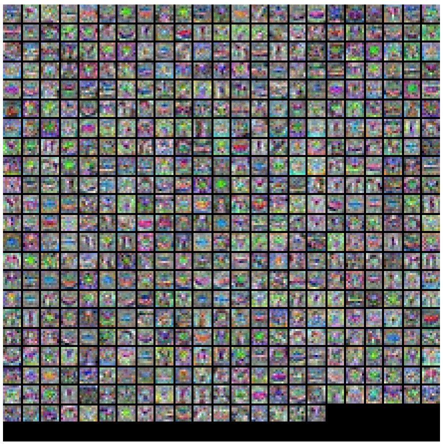
In [ ]:

```python
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Validation accuracy:  0.501
```

In [ ]:

```
# 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 [ ]:

```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.497
```

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.

$\color{blue}{\textit{Your Answer :}}$ 1,3 $\color{blue}{\textit{Your Explanation :}}$

The testing accuracy is lower than the training accuracy indicates that the model is overfitting in the dataset.

1. Training on a larger dataset can increase the diversity of the data, which may reduce the overfitting.
2. Too complex model may increase the overfitting.
3. Increasing the regularization strength avoid some weight with large magnitude, which avoid overfitting.