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.

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 [3]:

```
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 regularization loss.

In [4]:

```
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 [5]:

```
from cs231n.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[param_name])))
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-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 0.02.

In [6]:

```
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:  0.017149607938732093
```

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.

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 [8]:

```
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.

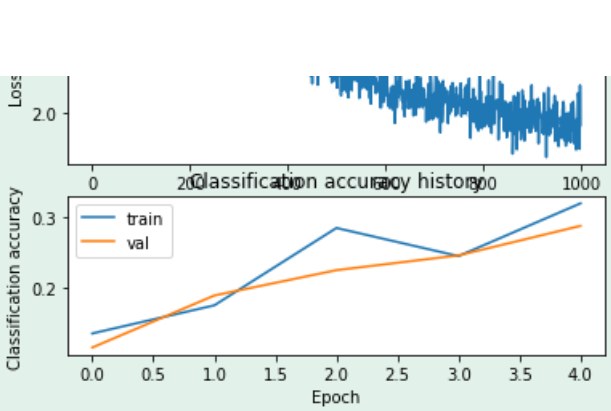
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [9]:

```
# 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()
```



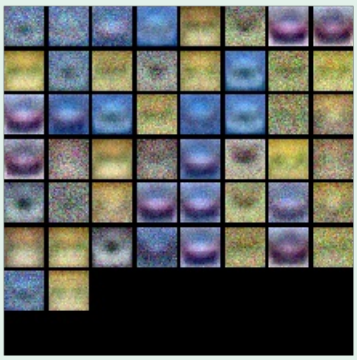
In [10]:

```
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.

YourAnswer: I will try different combinations of hidden size, learning rate, regulation strength and batch size to find the appropriate hyperparameters.

In [13]:

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

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

best_acc = -1

input_size = 32 * 32 * 3
num_classes = 10

hidden_sizes = [50, 100]
learning_rates = [1e-3, 1e-4]
regulation_strengths = [0.25, 0.5]
batch_sizes = [200, 400]

for hidden_size in hidden_sizes:
    for rate in learning_rates:
        for strength in regulation_strengths:
            for batch_sz in batch_sizes:
                # Initialize a new network
                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=batch_sz,
                                  learning_rate=rate, learning_rate_decay=0.95,
                                  reg=strength, verbose=True)

                val_acc = (net.predict(X_val) == y_val).mean()

                print ('hidden size = %d, learning rate = %e, regulation strength = %e, batch_size
= %d, Valid_accuracy: %f'
                       %(hidden_size, rate, strength, batch_sz, val_acc))

                if val_acc > best_acc:
                    best_acc = val_acc
                    best_net = net




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

```
iteration 0 / 1000: loss 2.302938
iteration 100 / 1000: loss 2.001446
iteration 200 / 1000: loss 1.805014
iteration 300 / 1000: loss 1.692433
iteration 400 / 1000: loss 1.608849
iteration 500 / 1000: loss 1.619214
iteration 600 / 1000: loss 1.513565
iteration 700 / 1000: loss 1.816291
iteration 800 / 1000: loss 1.462572
iteration 900 / 1000: loss 1.499910
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size = 2
00, Valid_accuracy: 0.455000
iteration 0 / 1000: loss 2.302957
iteration 100 / 1000: loss 1.970983
iteration 200 / 1000: loss 1.760164
iteration 300 / 1000: loss 1.747722
iteration 400 / 1000: loss 1.611945
iteration 500 / 1000: loss 1.537459
iteration 600 / 1000: loss 1.506438
iteration 700 / 1000: loss 1.542277
iteration 800 / 1000: loss 1.529512
iteration 900 / 1000: loss 1.556933
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size = 4
00, Valid_accuracy: 0.483000
```

```
iteration 0 / 1000: loss 2.303354
iteration 100 / 1000: loss 1.884682
iteration 200 / 1000: loss 1.850019
iteration 300 / 1000: loss 1.746901
iteration 400 / 1000: loss 1.687540
iteration 500 / 1000: loss 1.803003
iteration 600 / 1000: loss 1.607671
iteration 700 / 1000: loss 1.563636
iteration 800 / 1000: loss 1.547632
iteration 900 / 1000: loss 1.558945
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size = 2
00, Valid_accuracy: 0.471000
iteration 0 / 1000: loss 2.303356
iteration 100 / 1000: loss 2.021878
iteration 200 / 1000: loss 1.782859
iteration 300 / 1000: loss 1.703777
iteration 400 / 1000: loss 1.655029
iteration 500 / 1000: loss 1.622439
iteration 600 / 1000: loss 1.661049
iteration 700 / 1000: loss 1.550663
iteration 800 / 1000: loss 1.557937
iteration 900 / 1000: loss 1.572061
hidden size = 50, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size = 4
00, Valid_accuracy: 0.468000
iteration 0 / 1000: loss 2.302958
iteration 100 / 1000: loss 2.302256
iteration 200 / 1000: loss 2.298498
iteration 300 / 1000: loss 2.260528
iteration 400 / 1000: loss 2.176528
iteration 500 / 1000: loss 2.163990
iteration 600 / 1000: loss 2.009059
iteration 700 / 1000: loss 2.048202
iteration 800 / 1000: loss 1.959462
iteration 900 / 1000: loss 2.070273
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size = 2
00, Valid_accuracy: 0.282000
iteration 0 / 1000: loss 2.302967
iteration 100 / 1000: loss 2.302552
iteration 200 / 1000: loss 2.298638
iteration 300 / 1000: loss 2.274630
iteration 400 / 1000: loss 2.197956
iteration 500 / 1000: loss 2.177668
iteration 600 / 1000: loss 2.053459
iteration 700 / 1000: loss 2.077726
iteration 800 / 1000: loss 2.005571
iteration 900 / 1000: loss 1.967835
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size = 4
00, Valid_accuracy: 0.283000
iteration 0 / 1000: loss 2.303389
iteration 100 / 1000: loss 2.303012
iteration 200 / 1000: loss 2.300792
iteration 300 / 1000: loss 2.279080
iteration 400 / 1000: loss 2.206246
iteration 500 / 1000: loss 2.102063
iteration 600 / 1000: loss 2.087562
iteration 700 / 1000: loss 2.007441
iteration 800 / 1000: loss 2.022864
iteration 900 / 1000: loss 2.090471
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size = 2
00, Valid_accuracy: 0.275000
iteration 0 / 1000: loss 2.303373
iteration 100 / 1000: loss 2.302982
iteration 200 / 1000: loss 2.298774
iteration 300 / 1000: loss 2.259631
iteration 400 / 1000: loss 2.216077
iteration 500 / 1000: loss 2.103669
iteration 600 / 1000: loss 2.107711
iteration 700 / 1000: loss 2.030601
iteration 800 / 1000: loss 2.016085
iteration 900 / 1000: loss 1.986063
hidden size = 50, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size = 4
00, Valid_accuracy: 0.280000
iteration 0 / 1000: loss 2.303338
iteration 100 / 1000: loss 1.911545
iteration 200 / 1000: loss 1.683111
iteration 300 / 1000: loss 1.666524
iteration 400 / 1000: loss 1.609624
```

```
iteration 500 / 1000: loss 1.615796
iteration 600 / 1000: loss 1.507984
iteration 700 / 1000: loss 1.513616
iteration 800 / 1000: loss 1.433492
iteration 900 / 1000: loss 1.593103
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size =
200, Valid_accuracy: 0.481000
iteration 0 / 1000: loss 2.303353
iteration 100 / 1000: loss 1.988648
iteration 200 / 1000: loss 1.750819
iteration 300 / 1000: loss 1.689663
iteration 400 / 1000: loss 1.568600
iteration 500 / 1000: loss 1.646333
iteration 600 / 1000: loss 1.453001
iteration 700 / 1000: loss 1.543148
iteration 800 / 1000: loss 1.474791
iteration 900 / 1000: loss 1.434732
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 2.500000e-01, batch_size =
400, Valid_accuracy: 0.493000
iteration 0 / 1000: loss 2.304104
iteration 100 / 1000: loss 1.992265
iteration 200 / 1000: loss 1.790595
iteration 300 / 1000: loss 1.683221
iteration 400 / 1000: loss 1.630720
iteration 500 / 1000: loss 1.694728
iteration 600 / 1000: loss 1.553691
iteration 700 / 1000: loss 1.627905
iteration 800 / 1000: loss 1.565421
iteration 900 / 1000: loss 1.665791
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size =
200, Valid_accuracy: 0.463000
iteration 0 / 1000: loss 2.304139
iteration 100 / 1000: loss 1.914435
iteration 200 / 1000: loss 1.812060
iteration 300 / 1000: loss 1.688253
iteration 400 / 1000: loss 1.691924
iteration 500 / 1000: loss 1.630675
iteration 600 / 1000: loss 1.515584
iteration 700 / 1000: loss 1.580681
iteration 800 / 1000: loss 1.561937
iteration 900 / 1000: loss 1.449985
hidden size = 100, learning rate = 1.000000e-03, regulation strength = 5.000000e-01, batch_size =
400, Valid_accuracy: 0.487000
iteration 0 / 1000: loss 2.303320
iteration 100 / 1000: loss 2.301863
iteration 200 / 1000: loss 2.287207
iteration 300 / 1000: loss 2.234180
iteration 400 / 1000: loss 2.149615
iteration 500 / 1000: loss 2.142055
iteration 600 / 1000: loss 2.063547
iteration 700 / 1000: loss 2.073209
iteration 800 / 1000: loss 2.025929
iteration 900 / 1000: loss 2.002381
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size =
200, Valid_accuracy: 0.285000
iteration 0 / 1000: loss 2.303378
iteration 100 / 1000: loss 2.302502
iteration 200 / 1000: loss 2.292945
iteration 300 / 1000: loss 2.241410
iteration 400 / 1000: loss 2.161096
iteration 500 / 1000: loss 2.127070
iteration 600 / 1000: loss 2.055491
iteration 700 / 1000: loss 2.015337
iteration 800 / 1000: loss 2.030374
iteration 900 / 1000: loss 1.987144
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 2.500000e-01, batch_size =
400, Valid_accuracy: 0.291000
iteration 0 / 1000: loss 2.304117
iteration 100 / 1000: loss 2.303297
iteration 200 / 1000: loss 2.296294
iteration 300 / 1000: loss 2.257085
iteration 400 / 1000: loss 2.180464
iteration 500 / 1000: loss 2.089909
iteration 600 / 1000: loss 2.109783
iteration 700 / 1000: loss 2.042034
iteration 800 / 1000: loss 2.026012
iteration 900 / 1000: loss 1.958808
```

```
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size =
200, Valid_accuracy: 0.291000
iteration 0 / 1000: loss 2.304110
iteration 100 / 1000: loss 2.302891
iteration 200 / 1000: loss 2.294615
iteration 300 / 1000: loss 2.234861
iteration 400 / 1000: loss 2.171531
iteration 500 / 1000: loss 2.136451
iteration 600 / 1000: loss 2.038270
iteration 700 / 1000: loss 1.996320
iteration 800 / 1000: loss 2.046172
iteration 900 / 1000: loss 1.956266
hidden size = 100, learning rate = 1.000000e-04, regulation strength = 5.000000e-01, batch_size =
400, Valid_accuracy: 0.297000
```

In [15]:

```
# 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:  0.481
```

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.

YourAnswer: 1

YourExplanation:

1. Correct. Training on a larger dataset can make the model become more general and decrease the gap.
2. Incorrect. The low testing accuracy may due to overfitting. Adding more hidden units may still lead to this problem.
3. Incorrect. From the process of tuning hyperparameters we can know that increasing the regulation strength may lead to lower accuracy.