CNN-Layers

February 25, 2021

0.1 Convolutional neural network layers

In this notebook, we will build the convolutional neural network layers. This will be followed by a spatial batchnorm, and then in the final notebook of this assignment, we will train a CNN to further improve the validation accuracy on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

[1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%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))))
```

0.2 Implementing CNN layers

Just as we implemented modular layers for fully connected networks, batch normalization, and dropout, we'll want to implement modular layers for convolutional neural networks. These layers are in `nndl/conv_layers.py`.

0.2.1 Convolutional forward pass

Begin by implementing a naive version of the forward pass of the CNN that uses `for` loops. This function is `conv_forward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a triple `for` loop.

After you implement `conv_forward_naive`, test your implementation by running the cell below.

[2]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])
# Compare your output to ours; difference should be around 1e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

0.2.2 Convolutional backward pass

Now, implement a naive version of the backward pass of the CNN. The function is `conv_backward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a quadruple `for` loop.

After you implement `conv_backward_naive`, test your implementation by running the cell below.

[3]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_forward_naive(x,w,b,conv_param)

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around 1e-9'
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.366859820551726e-09
dw error:  2.783082080067301e-10
db error:  1.3164225673775657e-11
```

0.2.3 Max pool forward pass

In this section, we will implement the forward pass of the max pool. The function is `max_pool_forward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of implementation.

After you implement `max_pool_forward_naive`, test your implementation by running the cell below.

[4]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
```

```
                        [[-0.02736842, -0.01263158],
                         [ 0.03157895,  0.04631579]]],
                       [[[ 0.09052632,  0.10526316],
                         [ 0.14947368,  0.16421053]],
                        [[ 0.20842105,  0.22315789],
                         [ 0.26736842,  0.28210526]],
                        [[ 0.32631579,  0.34105263],
                         [ 0.38526316,  0.4       ]]]])
# Compare your output with ours. Difference should be around 1e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

0.2.4 Max pool backward pass

In this section, you will implement the backward pass of the max pool. The function is `max_pool_backward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of implementation.

After you implement `max_pool_backward_naive`, test your implementation by running the cell below.

[5]:
```
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x,␣
 ↪pool_param)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be around 1e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.275624396386197e-12
```

0.3 Fast implementation of the CNN layers

Implementing fast versions of the CNN layers can be difficult. We will provide you with the fast layers implemented by cs231n. They are provided in `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the cell below.

You should see pretty drastic speedups in the implementation of these layers. On our machine, the forward pass speeds up by 17x and the backward pass speeds up by 840x. Of course, these numbers will vary from machine to machine, as well as on your precise implementation of the naive layers.

[6]:
```
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 5.433432s
Fast: 0.009385s
```

```
Speedup: 578.927067x
Difference:  8.036182519306021e-11

Testing conv_backward_fast:
Naive: 7.733207s
Fast: 0.010346s
Speedup: 747.463267x
dx difference:  5.59985298544592e-12
dw difference:  3.4383578474379873e-13
db difference:  0.0
```

[7]:
```
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.399669s
fast: 0.002412s
speedup: 165.694672x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.570978s
```

```
speedup: 55.275262x
dx difference:  0.0
```

0.4 Implementation of cascaded layers

We've provided the following functions in `nndl/conv_layer_utils.py`: - conv_relu_forward - conv_relu_backward - conv_relu_pool_forward - conv_relu_pool_backward

These use the fast implementations of the conv net layers. You can test them below:

[8]:
```
from nndl.conv_layer_utils import conv_relu_pool_forward,␣
↪conv_relu_pool_backward

x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w,␣
↪b, conv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w,␣
↪b, conv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w,␣
↪b, conv_param, pool_param)[0], b, dout)

print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error:  7.23561848128953e-09
dw error:  2.2246344949353818e-09
db error:  5.056123476114607e-11
```

[9]:
```
from nndl.conv_layer_utils import conv_relu_forward, conv_relu_backward

x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
```

```
out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], b, dout)

print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu:
dx error:  2.0561215176446717e-09
dw error:  8.162001057204171e-10
db error:  6.808266080680872e-12
```

0.5 What next?

We saw how helpful batch normalization was for training FC nets. In the next notebook, we'll implement a batch normalization for convolutional neural networks, and then finish off by implementing a CNN to improve our validation accuracy on CIFAR-10.

CNN-BatchNorm

February 25, 2021

0.1 Spatial batch normalization

In fully connected networks, we performed batch normalization on the activations. To do something equivalent on CNNs, we modify batch normalization slightly.

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization accepts inputs of shape `(N, C, H, W)` and produces outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

How do we calculate the spatial averages? First, notice that for the `C` feature maps we have (i.e., the layer has `C` filters) that each of these ought to have its own batch norm statistics, since each feature map may be picking out very different features in the images. However, within a feature map, we may assume that across all inputs and across all locations in the feature map, there ought to be relatively similar first and second order statistics. Hence, one way to think of spatial batch-normalization is to reshape the `(N, C, H, W)` array as an `(N*H*W, C)` array and perform batch normalization on this array.

Since spatial batch norm and batch normalization are similar, it'd be good to at this point also copy and paste our prior implemented layers from HW #4. Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

You'll want to copy and paste from HW #4: - layers.py for your FC network layers, as well as batchnorm and dropout. - layer_utils.py for your combined FC network layers. - optim.py for your optimizers.

Be sure to place these in the `nndl/` directory so they're imported correctly. Note, as announced in class, we will not be releasing our solutions.

If you use your prior implementations of the batchnorm, then your spatial batchnorm implementation may be very short. Our implementations of the forward and backward pass are each 6 lines of code.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

[1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%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))))
```

0.2 Spatial batch normalization forward pass

Implement the forward pass, `spatial_batchnorm_forward` in `nndl/conv_layers.py`. Test your implementation by running the cell below.

[2]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
```

```
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 8.10916    10.38816015 10.52405417]
  Stds:  [4.20656815 4.50047185 4.29562312]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [-0.35790471  0.16270847  0.19519623]
  Stds:  [0.95113746 0.99783702 0.95235807]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [5.39374285 7.67810537 7.92815178]
  Stds:  [3.78506334 4.18366807 4.03346447]
```

0.3 Spatial batch normalization backward pass

Implement the backward pass, `spatial_batchnorm_backward` in `nndl/conv_layers.py`. Test your implementation by running the cell below.

[3]:
```
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
```

```
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  2.7990987144348948e-08
dgamma error:  1.2764557559882658e-11
dbeta error:  5.523504969109281e-12
```

[ ]:

CNN

February 25, 2021

1 Convolutional neural networks

In this notebook, we'll put together our convolutional layers to implement a 3-layer CNN. Then, we'll ask you to implement a CNN that can achieve $> 65\%$ validation error on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

If you have not completed the Spatial BatchNorm Notebook, please see the following description from that notebook:

Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their layer implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

You'll want to copy and paste from HW #4: - layers.py for your FC network layers, as well as batchnorm and dropout. - layer_utils.py for your combined FC network layers. - optim.py for your optimizers.

Be sure to place these in the `nndl/` directory so they're imported correctly. Note, as announced in class, we will not be releasing our solutions.

[1]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array,␣
↪eval_numerical_gradient
from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver
```

```
%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))))
```

[2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

1.1 Three layer CNN

In this notebook, you will implement a three layer CNN. The `ThreeLayerConvNet` class is in `nndl/cnn.py`. You'll need to modify that code for this section, including the initialization, as well as the calculation of the loss and gradients. You should be able to use the building blocks you have either earlier coded or that we have provided. Be sure to use the fast layers.

The architecture of this CNN will be:

conv - relu - 2x2 max pool - affine - relu - affine - softmax

We won't use batchnorm yet. You've also done enough of these to know how to debug; use the cells below.

Note: As we are implementing several layers CNN networks. The gradient error can be expected for the `eval_numerical_gradient()` function. If your `W1 max relative error` and `W2 max relative error` are around or below 0.01, they should be acceptable. Other errors should be less than 1e-5.

[10]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
```

```
num_classes = 10
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
 ↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('{} max relative error: {}'.format(param_name,␣
 ↪rel_error(param_grad_num, grads[param_name])))
```

```
W1 max relative error: 0.0018143911619591597
W2 max relative error: 0.004908117111230393
W3 max relative error: 2.613140536874486e-05
b1 max relative error: 3.284939731567521e-05
b2 max relative error: 8.58347419303169e-07
b3 max relative error: 1.3538900116320964e-09
```

1.1.1 Overfit small dataset

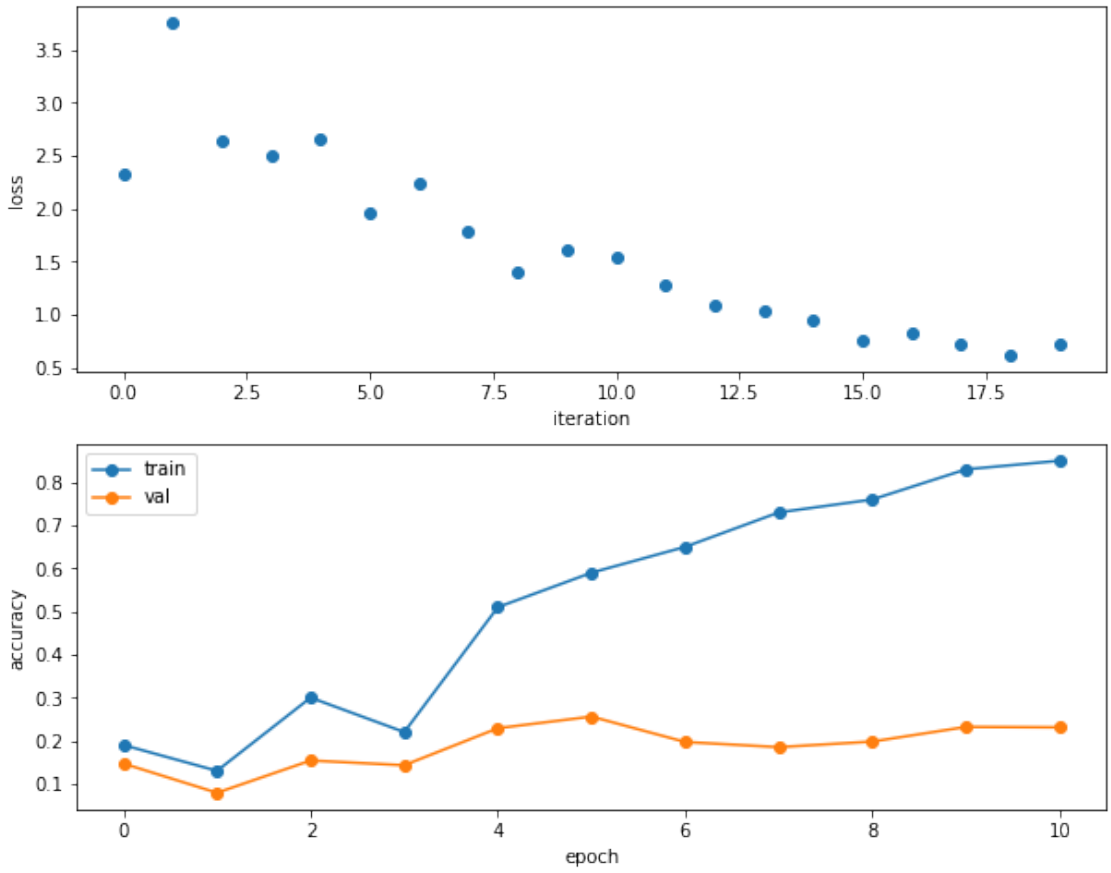
To check your CNN implementation, let's overfit a small dataset.

[11]:
```
num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 20) loss: 2.326105
(Epoch 0 / 10) train acc: 0.190000; val_acc: 0.147000
```

```
(Iteration 2 / 20) loss: 3.760770
(Epoch 1 / 10) train acc: 0.130000; val_acc: 0.079000
(Iteration 3 / 20) loss: 2.643307
(Iteration 4 / 20) loss: 2.497411
(Epoch 2 / 10) train acc: 0.300000; val_acc: 0.154000
(Iteration 5 / 20) loss: 2.651551
(Iteration 6 / 20) loss: 1.961063
(Epoch 3 / 10) train acc: 0.220000; val_acc: 0.143000
(Iteration 7 / 20) loss: 2.236175
(Iteration 8 / 20) loss: 1.783043
(Epoch 4 / 10) train acc: 0.510000; val_acc: 0.229000
(Iteration 9 / 20) loss: 1.398427
(Iteration 10 / 20) loss: 1.611017
(Epoch 5 / 10) train acc: 0.590000; val_acc: 0.256000
(Iteration 11 / 20) loss: 1.535831
(Iteration 12 / 20) loss: 1.280258
(Epoch 6 / 10) train acc: 0.650000; val_acc: 0.197000
(Iteration 13 / 20) loss: 1.084481
(Iteration 14 / 20) loss: 1.040968
(Epoch 7 / 10) train acc: 0.730000; val_acc: 0.185000
(Iteration 15 / 20) loss: 0.949079
(Iteration 16 / 20) loss: 0.753528
(Epoch 8 / 10) train acc: 0.760000; val_acc: 0.198000
(Iteration 17 / 20) loss: 0.833763
(Iteration 18 / 20) loss: 0.713428
(Epoch 9 / 10) train acc: 0.830000; val_acc: 0.232000
(Iteration 19 / 20) loss: 0.623954
(Iteration 20 / 20) loss: 0.727199
(Epoch 10 / 10) train acc: 0.850000; val_acc: 0.231000
```

[12]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



1.2 Train the network

Now we train the 3 layer CNN on CIFAR-10 and assess its accuracy.

[13]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.303649
(Epoch 0 / 1) train acc: 0.113000; val_acc: 0.105000
(Iteration 21 / 980) loss: 1.933823
(Iteration 41 / 980) loss: 2.357085
(Iteration 61 / 980) loss: 1.943011
```

```
(Iteration 81 / 980) loss: 2.093806
(Iteration 101 / 980) loss: 1.754699
(Iteration 121 / 980) loss: 1.655813
(Iteration 141 / 980) loss: 1.614959
(Iteration 161 / 980) loss: 1.620232
(Iteration 181 / 980) loss: 1.527127
(Iteration 201 / 980) loss: 1.724129
(Iteration 221 / 980) loss: 1.864630
(Iteration 241 / 980) loss: 1.792950
(Iteration 261 / 980) loss: 1.536456
(Iteration 281 / 980) loss: 1.736306
(Iteration 301 / 980) loss: 1.899632
(Iteration 321 / 980) loss: 1.525947
(Iteration 341 / 980) loss: 1.766815
(Iteration 361 / 980) loss: 1.676856
(Iteration 381 / 980) loss: 1.535262
(Iteration 401 / 980) loss: 1.592934
(Iteration 421 / 980) loss: 1.769800
(Iteration 441 / 980) loss: 1.523294
(Iteration 461 / 980) loss: 1.493456
(Iteration 481 / 980) loss: 1.572871
(Iteration 501 / 980) loss: 1.732399
(Iteration 521 / 980) loss: 1.395844
(Iteration 541 / 980) loss: 1.869968
(Iteration 561 / 980) loss: 1.571103
(Iteration 581 / 980) loss: 1.600936
(Iteration 601 / 980) loss: 1.861201
(Iteration 621 / 980) loss: 1.624602
(Iteration 641 / 980) loss: 1.595063
(Iteration 661 / 980) loss: 1.615744
(Iteration 681 / 980) loss: 1.322430
(Iteration 701 / 980) loss: 1.496684
(Iteration 721 / 980) loss: 1.587285
(Iteration 741 / 980) loss: 1.479097
(Iteration 761 / 980) loss: 1.414258
(Iteration 781 / 980) loss: 1.563884
(Iteration 801 / 980) loss: 1.537205
(Iteration 821 / 980) loss: 1.631709
(Iteration 841 / 980) loss: 1.588504
(Iteration 861 / 980) loss: 1.373891
(Iteration 881 / 980) loss: 1.683461
(Iteration 901 / 980) loss: 1.276985
(Iteration 921 / 980) loss: 1.215760
(Iteration 941 / 980) loss: 1.600567
(Iteration 961 / 980) loss: 1.720523
(Epoch 1 / 1) train acc: 0.471000; val_acc: 0.491000
```

2 Get > 65% validation accuracy on CIFAR-10.

In the last part of the assignment, we'll now ask you to train a CNN to get better than 65% validation accuracy on CIFAR-10.

2.0.1 Things you should try:

- Filter size: Above we used 7x7; but VGGNet and onwards showed stacks of 3x3 filters are good.

- Number of filters: Above we used 32 filters. Do more or fewer do better?
- Batch normalization: Try adding spatial batch normalization after convolution layers and vanilla batch normalization aafter affine layers. Do your networks train faster?
- Network architecture: Can a deeper CNN do better? Consider these architectures:
  - [conv-relu-pool]xN - conv - relu - [affine]xM - [softmax or SVM]
  - [conv-relu-pool]XN - [affine]XM - [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN - [affine]xM - [softmax or SVM]

2.0.2 Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

[8]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #
model = ThreeLayerConvNet(
    weight_scale=0.001,
    hidden_dim=500,
    reg=0.005,
    filter_size=3,
    num_filters=64,
    use_batchnorm=True
)

solver = Solver(model, data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
```

```
                  'learning_rate': 1e-3,
                },
                lr_decay = 0.99,
                verbose=True, print_every=100)
solver.train()
# ============================================================== #
# END YOUR CODE HERE
# ============================================================== #
```

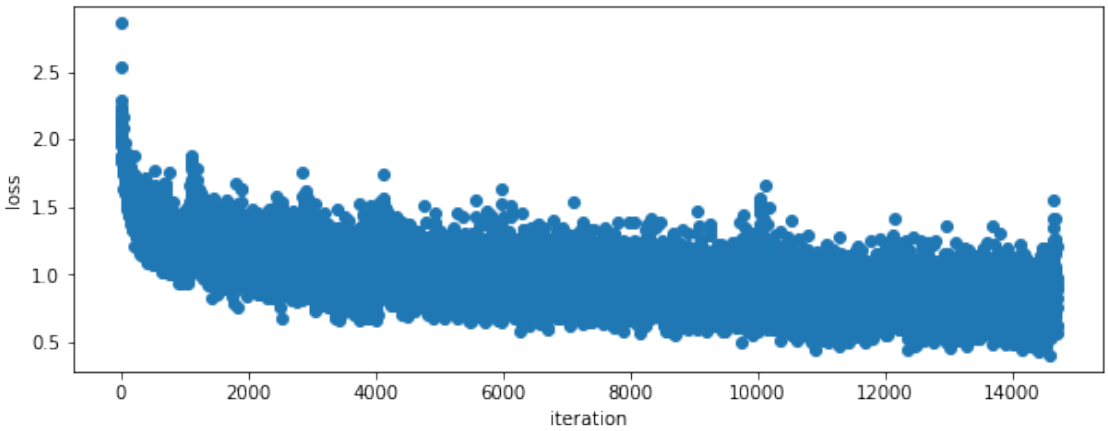
```
(Iteration 1 / 14700) loss: 2.866413
(Epoch 0 / 15) train acc: 0.170000; val_acc: 0.180000
(Iteration 101 / 14700) loss: 1.648682
(Iteration 201 / 14700) loss: 1.557947
(Iteration 301 / 14700) loss: 1.293185
(Iteration 401 / 14700) loss: 1.394603
(Iteration 501 / 14700) loss: 1.482404
(Iteration 601 / 14700) loss: 1.131207
(Iteration 701 / 14700) loss: 1.534966
(Iteration 801 / 14700) loss: 1.168253
(Iteration 901 / 14700) loss: 1.000626
(Epoch 1 / 15) train acc: 0.598000; val_acc: 0.554000
(Iteration 1001 / 14700) loss: 1.049128
(Iteration 1101 / 14700) loss: 1.512897
(Iteration 1201 / 14700) loss: 1.424432
(Iteration 1301 / 14700) loss: 1.035193
(Iteration 1401 / 14700) loss: 0.978838
(Iteration 1501 / 14700) loss: 1.221826
(Iteration 1601 / 14700) loss: 1.317455
(Iteration 1701 / 14700) loss: 1.249326
(Iteration 1801 / 14700) loss: 1.036787
(Iteration 1901 / 14700) loss: 1.183147
(Epoch 2 / 15) train acc: 0.636000; val_acc: 0.609000
(Iteration 2001 / 14700) loss: 1.087472
(Iteration 2101 / 14700) loss: 1.010321
(Iteration 2201 / 14700) loss: 1.127407
(Iteration 2301 / 14700) loss: 1.104468
(Iteration 2401 / 14700) loss: 1.044643
(Iteration 2501 / 14700) loss: 1.243717
(Iteration 2601 / 14700) loss: 1.069955
(Iteration 2701 / 14700) loss: 1.018863
(Iteration 2801 / 14700) loss: 0.969890
(Iteration 2901 / 14700) loss: 1.163248
(Epoch 3 / 15) train acc: 0.643000; val_acc: 0.579000
(Iteration 3001 / 14700) loss: 0.917717
(Iteration 3101 / 14700) loss: 0.998466
(Iteration 3201 / 14700) loss: 1.061334
(Iteration 3301 / 14700) loss: 1.158182
```

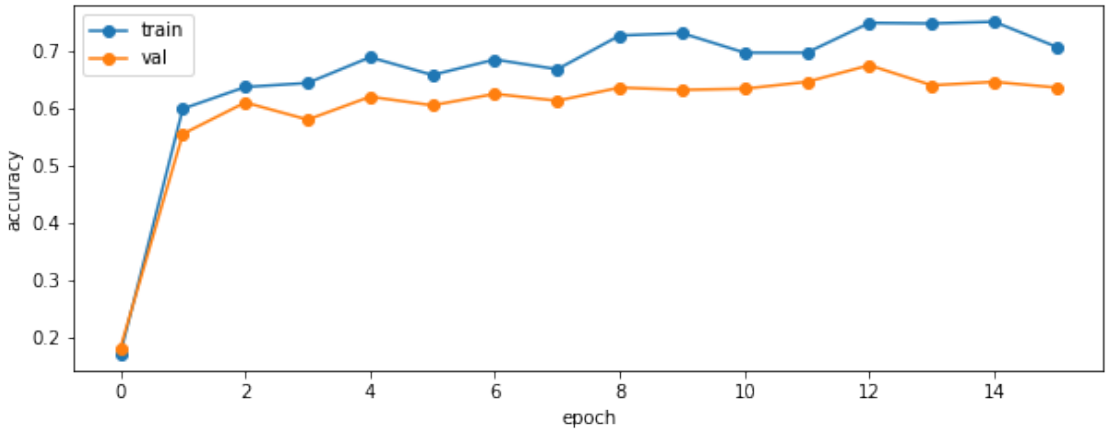
```
(Iteration 3401 / 14700) loss: 0.949251
(Iteration 3501 / 14700) loss: 1.029499
(Iteration 3601 / 14700) loss: 0.845543
(Iteration 3701 / 14700) loss: 1.041103
(Iteration 3801 / 14700) loss: 1.064841
(Iteration 3901 / 14700) loss: 0.701562
(Epoch 4 / 15) train acc: 0.688000; val_acc: 0.619000
(Iteration 4001 / 14700) loss: 1.082019
(Iteration 4101 / 14700) loss: 0.992090
(Iteration 4201 / 14700) loss: 0.904017
(Iteration 4301 / 14700) loss: 1.197572
(Iteration 4401 / 14700) loss: 1.204432
(Iteration 4501 / 14700) loss: 0.969512
(Iteration 4601 / 14700) loss: 1.120841
(Iteration 4701 / 14700) loss: 0.854697
(Iteration 4801 / 14700) loss: 1.117440
(Epoch 5 / 15) train acc: 0.657000; val_acc: 0.604000
(Iteration 4901 / 14700) loss: 1.007085
(Iteration 5001 / 14700) loss: 1.000993
(Iteration 5101 / 14700) loss: 1.050393
(Iteration 5201 / 14700) loss: 0.990949
(Iteration 5301 / 14700) loss: 1.231830
(Iteration 5401 / 14700) loss: 1.241174
(Iteration 5501 / 14700) loss: 0.953256
(Iteration 5601 / 14700) loss: 1.020885
(Iteration 5701 / 14700) loss: 1.049284
(Iteration 5801 / 14700) loss: 0.811337
(Epoch 6 / 15) train acc: 0.684000; val_acc: 0.624000
(Iteration 5901 / 14700) loss: 0.885619
(Iteration 6001 / 14700) loss: 1.011022
(Iteration 6101 / 14700) loss: 0.905470
(Iteration 6201 / 14700) loss: 1.000364
(Iteration 6301 / 14700) loss: 0.800229
(Iteration 6401 / 14700) loss: 0.946079
(Iteration 6501 / 14700) loss: 0.848335
(Iteration 6601 / 14700) loss: 0.947571
(Iteration 6701 / 14700) loss: 0.934981
(Iteration 6801 / 14700) loss: 0.965145
(Epoch 7 / 15) train acc: 0.667000; val_acc: 0.612000
(Iteration 6901 / 14700) loss: 1.154362
(Iteration 7001 / 14700) loss: 1.098023
(Iteration 7101 / 14700) loss: 1.296176
(Iteration 7201 / 14700) loss: 1.131676
(Iteration 7301 / 14700) loss: 1.036344
(Iteration 7401 / 14700) loss: 0.764804
(Iteration 7501 / 14700) loss: 1.102464
(Iteration 7601 / 14700) loss: 0.987521
(Iteration 7701 / 14700) loss: 0.953823
```

```
(Iteration 7801 / 14700) loss: 1.010395
(Epoch 8 / 15) train acc: 0.726000; val_acc: 0.635000
(Iteration 7901 / 14700) loss: 1.018442
(Iteration 8001 / 14700) loss: 0.890742
(Iteration 8101 / 14700) loss: 0.874286
(Iteration 8201 / 14700) loss: 0.995034
(Iteration 8301 / 14700) loss: 0.825760
(Iteration 8401 / 14700) loss: 0.746960
(Iteration 8501 / 14700) loss: 0.817462
(Iteration 8601 / 14700) loss: 0.940936
(Iteration 8701 / 14700) loss: 1.089417
(Iteration 8801 / 14700) loss: 0.769532
(Epoch 9 / 15) train acc: 0.730000; val_acc: 0.631000
(Iteration 8901 / 14700) loss: 0.859279
(Iteration 9001 / 14700) loss: 0.859929
(Iteration 9101 / 14700) loss: 1.155940
(Iteration 9201 / 14700) loss: 0.832026
(Iteration 9301 / 14700) loss: 1.183165
(Iteration 9401 / 14700) loss: 0.834001
(Iteration 9501 / 14700) loss: 0.874388
(Iteration 9601 / 14700) loss: 1.082936
(Iteration 9701 / 14700) loss: 0.802369
(Epoch 10 / 15) train acc: 0.696000; val_acc: 0.633000
(Iteration 9801 / 14700) loss: 0.845951
(Iteration 9901 / 14700) loss: 1.061316
(Iteration 10001 / 14700) loss: 0.727808
(Iteration 10101 / 14700) loss: 0.959739
(Iteration 10201 / 14700) loss: 0.916254
(Iteration 10301 / 14700) loss: 1.015313
(Iteration 10401 / 14700) loss: 0.773334
(Iteration 10501 / 14700) loss: 0.766654
(Iteration 10601 / 14700) loss: 0.542458
(Iteration 10701 / 14700) loss: 0.840477
(Epoch 11 / 15) train acc: 0.696000; val_acc: 0.645000
(Iteration 10801 / 14700) loss: 0.675197
(Iteration 10901 / 14700) loss: 0.734959
(Iteration 11001 / 14700) loss: 0.878776
(Iteration 11101 / 14700) loss: 0.934710
(Iteration 11201 / 14700) loss: 0.970829
(Iteration 11301 / 14700) loss: 0.754153
(Iteration 11401 / 14700) loss: 0.872738
(Iteration 11501 / 14700) loss: 1.050783
(Iteration 11601 / 14700) loss: 0.761401
(Iteration 11701 / 14700) loss: 0.603172
(Epoch 12 / 15) train acc: 0.748000; val_acc: 0.674000
(Iteration 11801 / 14700) loss: 1.077444
(Iteration 11901 / 14700) loss: 0.818656
(Iteration 12001 / 14700) loss: 1.049798
```

```
(Iteration 12101 / 14700) loss: 0.901774
(Iteration 12201 / 14700) loss: 0.873657
(Iteration 12301 / 14700) loss: 0.714776
(Iteration 12401 / 14700) loss: 0.747913
(Iteration 12501 / 14700) loss: 0.648796
(Iteration 12601 / 14700) loss: 0.829350
(Iteration 12701 / 14700) loss: 0.647658
(Epoch 13 / 15) train acc: 0.747000; val_acc: 0.639000
(Iteration 12801 / 14700) loss: 0.832330
(Iteration 12901 / 14700) loss: 0.674444
(Iteration 13001 / 14700) loss: 0.688189
(Iteration 13101 / 14700) loss: 0.743763
(Iteration 13201 / 14700) loss: 0.613485
(Iteration 13301 / 14700) loss: 0.656347
(Iteration 13401 / 14700) loss: 0.764100
(Iteration 13501 / 14700) loss: 0.629584
(Iteration 13601 / 14700) loss: 0.589967
(Iteration 13701 / 14700) loss: 0.934189
(Epoch 14 / 15) train acc: 0.750000; val_acc: 0.645000
(Iteration 13801 / 14700) loss: 0.662257
(Iteration 13901 / 14700) loss: 0.809934
(Iteration 14001 / 14700) loss: 0.582026
(Iteration 14101 / 14700) loss: 0.505915
(Iteration 14201 / 14700) loss: 0.839436
(Iteration 14301 / 14700) loss: 0.751480
(Iteration 14401 / 14700) loss: 0.611209
(Iteration 14501 / 14700) loss: 0.561000
(Iteration 14601 / 14700) loss: 0.406979
(Epoch 15 / 15) train acc: 0.706000; val_acc: 0.635000
```

[9]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```


loss
2.5
2.0
1.5
1.0
0.5
0
2000
4000
6000
8000
10000
12000
14000
iteration


train
val
accuracy
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
14
epoch

[ ]:

[ ]: