Margaret Capetz

147 HW 5

$CL_1 : m_1 \times m_1$
$CL_2 : m_2 \times m_2$
$CL_3 : m_3 \times m_3$
stride = 1

Receptive field: output depends on an $n \times n$ patch

1)

a) Receptive field of each neuron in CL_1

$\boxed{m_1 \times m_1}$, filter size

b) CL_2: $\boxed{(m_2 + m_1 - 1) \times (m_2 + m_1 - 1)}$

c) strides of s_1, s_2, s_3 on conv layers CL_1, CL_2, CL_3. How would this affect the receptive fields of CL_1, CL_2.

CL_1 receptive field remains unchanged.

CL_2 receptive field changes based on the stride of the prev layer, s_1.

CL_2 Receptive Field = $s_1(m_2 - 1) + m_1$

d) Generalized expression:

CL_i Receptive Field =

$$\boxed{CL_K = \left(\sum_{j=1}^{K} \left(\prod_{i=0}^{j-1} s_{i+1} \right) (m_j - 1) \right) + 1}$$

e) How to increase the receptive fields of neurons in CNNs?

i) Increase the stride.

ii) Stack more layers.

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.

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

import time

import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from utils.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))))
```

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`.

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.

In [ ]:
```
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
```

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.

In [ ]:
```
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)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param)

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:  4.043976113274569e-09
dw error:  4.037122047864299e-10
db error:  1.1120861980084216e-11
```

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.

In [ ]:
```
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
```

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.

In [ ]:
```
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

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:  3.2756420839091903e-12
```

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 utils. They are provided in `utils/fast_layers.py`.

The fast convolution implementation depends on a Cython extension ('pip install Cython' to your virtual environment); to compile it you need to run the following from the `utils` 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.

In [ ]:

```python
from utils.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: 0.216499s
Fast: 0.011922s
Speedup: 18.159444x
Difference:  1.006304546092691e-10

Testing conv_backward_fast:
Naive: 23.339216s
Fast: 0.006694s
Speedup: 3486.546533x
dx difference:  6.338266515357968e-12
dw difference:  2.3231859612709525e-13
db difference:  0.0
```

In [ ]:

```python
from utils.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: 0.086644s
fast: 0.003004s
speedup: 28.844511x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.460452s
speedup: 59.098351x
dx difference:  0.0
```

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:

In [ ]:
```
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_pa
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_pa
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_pa

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:  6.81956322975101e-09
dw error:  1.1003104453849371e-08
db error:  9.578922097276023e-11
```

In [ ]:
```
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)[
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param)[
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param)[

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:  1.4658621430932226e-09
dw error:  8.407279624028986e-10
db error:  3.8178792802034695e-11
```

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.

```
# conv layers updated

import numpy as np
from nndl.layers import *
import pdb


def conv_forward_naive(x, w, b, conv_param):
  """
  A naive implementation of the forward pass for a convolutional
layer.

  The input consists of N data points, each with C channels, height H
and width
  W. We convolve each input with F different filters, where each
filter spans
  all C channels and has height HH and width HH.

  Input:
  - x: Input data of shape (N, C, H, W)
  - w: Filter weights of shape (F, C, HH, WW)
  - b: Biases, of shape (F,)
  - conv_param: A dictionary with the following keys:
    - 'stride': The number of pixels between adjacent receptive fields
in the
      horizontal and vertical directions.
    - 'pad': The number of pixels that will be used to zero-pad the
input.

  Returns a tuple of:
  - out: Output data, of shape (N, F, H', W') where H' and W' are
given by
    H' = 1 + (H + 2 * pad - HH) / stride
    W' = 1 + (W + 2 * pad - WW) / stride
  - cache: (x, w, b, conv_param)
  """
  out = None
  pad = conv_param['pad']
  stride = conv_param['stride']

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the forward pass of a convolutional neural network.
  #   Store the output as 'out'.
  #   Hint: to pad the array, you can use the function np.pad.
  # ================================================================ #
  N, C, H, W = x.shape
  F, C, HH, WW = w.shape

  # check if valid conv
```

```
  assert (H + 2 * pad - HH) % stride == 0
  assert (W + 2 * pad - WW) % stride == 0

  # only pad the third and fourth axes
  x_pad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)),
mode='constant')

  # H_prime, W_prime are the output height and width
  H_prime = int(1 + (H + 2 * pad - HH) / stride)
  W_prime = int(1 + (W + 2 * pad - WW) / stride)

  out = np.zeros((N, F, H_prime, W_prime))

  for n in range(N): # number of batches
    for i in range(H_prime): # output height
      for j in range(W_prime): # output width
        seg = x_pad[n,:,i*stride:i*stride + HH, j*stride:j*stride +
WW]
        out[n,:,i,j] = np.sum(seg * w,axis=(1,2,3)) + b



  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

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


def conv_backward_naive(dout, cache):
  """
  A naive implementation of the backward pass for a convolutional
layer.

  Inputs:
  - dout: Upstream derivatives.
  - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

  Returns a tuple of:
  - dx: Gradient with respect to x
  - dw: Gradient with respect to w
  - db: Gradient with respect to b
  """
  dx, dw, db = None, None, None

  N, F, out_height, out_width = dout.shape
  x, w, b, conv_param = cache

  stride, pad = [conv_param['stride'], conv_param['pad']]
```

```
  xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)),
mode='constant')
  num_filts, _, f_height, f_width = w.shape

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the backward pass of a convolutional neural network.
  #   Calculate the gradients: dx, dw, and db.
  # ================================================================ #
  # inits
  dx = np.zeros_like(x)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)


  # dims
  N, C, H, W = x.shape
  F, _, HH, WW = w.shape
  _, _, H_out, W_out = dout.shape

  # db
  db = np.sum(dout, axis=(0, 2, 3))

  # dw
  for f in range(F): # looping through filters
      for c in range(C): # looping through channels
          for i in range(HH): # looping through weight height
              for j in range(WW): # looping through weight width
                  dw[f, c, i ,j] = np.sum(xpad[:,  c, i: i + H_out *
stride : stride, j : j + W_out* stride : stride] * dout[:, f, :, :])
                  # dw at determined segment
                  # np sum of (xpad matrix * dout matrix)

  # dx
  dx = np.zeros(x.shape)
  # loop through the number of examples
  for n in range(N): # hi, wi: loop through x
      for hx in range(H):
          for wx in range(W):
              y_indexes = [] # will contain valid indices of y
              w_indexes = [] # will contain valid indices of weight
(w)
              for i in range(H_out): # H_ is from dout
                  for j in range(W_out):  # W_ is from dout
                      # i, j: loop through output
                      # verify: is the range within weights limits?
                      h_range = (hx + pad - i * stride) # height range
                      w_range = (wx + pad - j * stride) # weight range
                      if (h_range >= 0) and (h_range < HH) and
(w_range >= 0) and (w_range < WW):
```

```
                        w_indexes.append((h_range, w_range))
                        y_indexes.append((i, j))

                for f in range(F): # filters loop
                    # windex_f and yindex_f from python zip of
w_indexes, y_indexes (as determined above)
                    # increment by np.sum ( w_matrix * dout_matrix) for
valid indices of y and weights
                    dx[n, : , hx, wx] += np.sum([w[f, :, windex_f[0],
windex_f[1]] * dout[n, f, yindex_f[0], yindex_f[1]] for windex_f,
yindex_f in zip(w_indexes, y_indexes)], 0)



  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return dx, dw, db


def max_pool_forward_naive(x, pool_param):
  """
  A naive implementation of the forward pass for a max pooling layer.

  Inputs:
  - x: Input data, of shape (N, C, H, W)
  - pool_param: dictionary with the following keys:
    - 'pool_height': The height of each pooling region
    - 'pool_width': The width of each pooling region
    - 'stride': The distance between adjacent pooling regions

  Returns a tuple of:
  - out: Output data
  - cache: (x, pool_param)
  """
  out = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling forward pass.
  # ================================================================ #

  N, C, H, W = x.shape
  HH = pool_param['pool_height']
  WW = pool_param['pool_width']
  stride = pool_param['stride']

  # H_prime, W_prime are the output height and width
  H_prime = int(1 + (H - HH) / stride)
```

```
  W_prime = int(1 + (W - WW) / stride)

  out = np.zeros((N, C, H_prime, W_prime))

  for n in range(N): # number of batches
    for i in range(H_prime): # output height
      for j in range(W_prime): # output width
        seg = x[n,:,i*stride:i*stride + HH, j*stride:j*stride + WW]
        out[n,:,i,j] = np.amax(seg,axis=(1,2))

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #
  cache = (x, pool_param)
  return out, cache

def max_pool_backward_naive(dout, cache):
  """
  A naive implementation of the backward pass for a max pooling layer.

  Inputs:
  - dout: Upstream derivatives
  - cache: A tuple of (x, pool_param) as in the forward pass.

  Returns:
  - dx: Gradient with respect to x
  """
  dx = None
  x, pool_param = cache
  pool_height, pool_width, stride = pool_param['pool_height'],
pool_param['pool_width'], pool_param['stride']

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling backward pass.
  # ================================================================ #

  N, C, H, W = x.shape
  N, C, dout_height, dout_width = dout.shape

  dx = np.zeros_like(x)

  for n in range(N):                              # loop over the
number of training samples
    for c in range(C):                          # loop over the number
of channels
        for i in range(dout_height):            # loop over vertical
axis of the dout
            for j in range(dout_width):         # loop over horizontal
axis of the dout
```

```
                i_, j_ = np.where(np.max(x[n, c, i * stride : i *
stride + pool_height, j * stride : j * stride + pool_width]) == x[n,
c, i * stride : i * stride + pool_height, j * stride : j * stride +
pool_width])
                i_, j_ = i_[0], j_[0]
                dx[n, c, i * stride : i * stride + pool_height, j *
stride : j * stride + pool_width][i_, j_] = dout[n, c, i, j]

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return dx

def spatial_batchnorm_forward(x, gamma, beta, bn_param):
  """
  Computes the forward pass for spatial batch normalization.

  Inputs:
  - x: Input data of shape (N, C, H, W)
  - gamma: Scale parameter, of shape (C,)
  - beta: Shift parameter, of shape (C,)
  - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance. momentum=0 means
that
      old information is discarded completely at every time step,
while
      momentum=1 means that new information is never incorporated. The
      default of momentum=0.9 should work well in most situations.
    - running_mean: Array of shape (D,) giving running mean of
features
    - running_var Array of shape (D,) giving running variance of
features

  Returns a tuple of:
  - out: Output data, of shape (N, C, H, W)
  - cache: Values needed for the backward pass
  """
  out, cache = None, None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the spatial batchnorm forward pass.
  #
  #   You may find it useful to use the batchnorm forward pass you
  #   implemented in HW #4.
  # ================================================================ #
```

```python
    N, C, W, H = x.shape
    xr = x.reshape(N*H*W, C)
    out, cache = batchnorm_forward(xr, gamma, beta, bn_param)
    out = out.reshape(N, C, W, H)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return out, cache


def spatial_batchnorm_backward(dout, cache):
    """
    Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the spatial batchnorm backward pass.
    #
    #   You may find it useful to use the batchnorm forward pass you
    #   implemented in HW #4.
    # ================================================================ #

    N, C, W, H = dout.shape

    doutr = dout.reshape(N*H*W, C)

    dx, dgamma, dbeta = batchnorm_backward(doutr, cache)

    dx = dx.reshape(N, C, W, H)
    dgamma = dgamma.reshape(C,)
    dbeta = dbeta.reshape(C,)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx, dgamma, dbeta
```

```
#layer utils

from .layers import *

def affine_relu_forward(x, w, b):
    """
    Convenience layer that performs an affine transform followed by a
ReLU

    Inputs:
    - x: Input to the affine layer
    - w, b: Weights for the affine layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, fc_cache = affine_forward(x, w, b)
    out, relu_cache = relu_forward(a)
    cache = (fc_cache, relu_cache)
    return out, cache


def affine_relu_backward(dout, cache):
    """
    Backward pass for the affine-relu convenience layer
    """
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db

def affine_batchnorm_relu_forward(x, w, b, gamma, beta, bn_params):
    """
    Convenience layer that performs an affine transform followed by a
ReLU

    Inputs:
    - x: Input to the affine layer
    - w, b: Weights for the affine layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, fc_cache = affine_forward(x, w, b)
    # print("a shape", a.shape)
    # print("gamma shape", gamma.shape)
    batchnorm, batch_cache = batchnorm_forward(a, gamma, beta,
bn_params)
```

```
    out, relu_cache = relu_forward(batchnorm)
    cache = (fc_cache, batch_cache, relu_cache)
    return out, cache



def affine_batchnorm_relu_backward(dout, cache):
    """
    Backward pass for the affine-relu convenience layer
    """
    fc_cache, batch_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    # print("relu bwd done")
    dx, dgamma, dbeta = batchnorm_backward(da, batch_cache)

    dx, dw, db = affine_backward(dx, fc_cache)
    # print("aff bwd done")
    return dx, dw, db, dgamma, dbeta
```

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.

In [ ]:
```python
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.conv_layers import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from utils.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))))
```

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.

In [ ]:

```
# 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:  [ 9.47400868 10.80047711  9.85082778]
  Stds:  [3.74823649 4.17972521 3.89743175]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [-0.14559759  0.17189589 -0.0262983 ]
  Stds:  [0.99086534 0.98576606 0.9975022 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6.2680601  7.91012751 6.82181238]
  Stds:  [3.7763465  4.461124   4.05484016]
```

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.

In [ ]:

```
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:  9.404491016687663e-09
dgamma error:  2.691810258180157e-11
dbeta error:  5.628067311374074e-12
```

In [ ]:

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.

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.

In [ ]:

```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from nndl.layers import *
from nndl.conv_layers import *
from utils.fast_layers import *
from utils.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))))
```

In [ ]:

```
# 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,)
```

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.

In [ ]:

```
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,
    e = rel_error(param_grad_num, grads[param_name])
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, grads
```

```
W1 max relative error: 0.00013717811956023954
W2 max relative error: 0.0004634868126112562
W3 max relative error: 3.88017805272466e-05
b1 max relative error: 2.1764014481345942e-05
b2 max relative error: 7.205694929161584e-08
b3 max relative error: 8.766444943918748e-10
```

Overfit small dataset

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

In [ ]:

```
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()
```

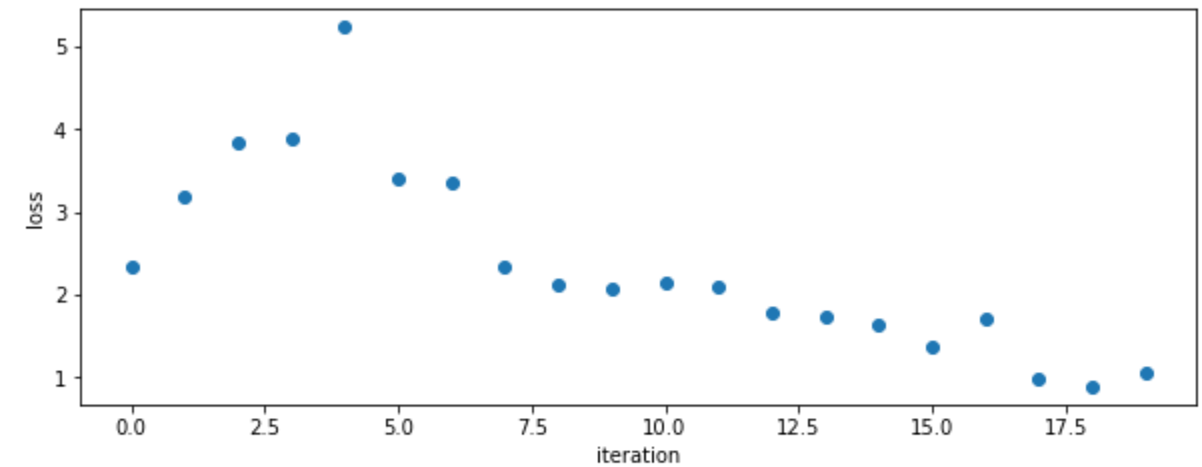
```
(Iteration 1 / 20) loss: 2.348549
(Epoch 0 / 10) train acc: 0.130000; val_acc: 0.109000
(Iteration 2 / 20) loss: 3.192955
(Epoch 1 / 10) train acc: 0.210000; val_acc: 0.118000
(Iteration 3 / 20) loss: 3.830671
(Iteration 4 / 20) loss: 3.893182
(Epoch 2 / 10) train acc: 0.110000; val_acc: 0.107000
(Iteration 5 / 20) loss: 5.242046
(Iteration 6 / 20) loss: 3.402059
(Epoch 3 / 10) train acc: 0.180000; val_acc: 0.137000
(Iteration 7 / 20) loss: 3.356559
(Iteration 8 / 20) loss: 2.349357
(Epoch 4 / 10) train acc: 0.220000; val_acc: 0.118000
(Iteration 9 / 20) loss: 2.113121
(Iteration 10 / 20) loss: 2.063497
(Epoch 5 / 10) train acc: 0.340000; val_acc: 0.168000
(Iteration 11 / 20) loss: 2.153304
(Iteration 12 / 20) loss: 2.087311
(Epoch 6 / 10) train acc: 0.420000; val_acc: 0.163000
(Iteration 13 / 20) loss: 1.772500
(Iteration 14 / 20) loss: 1.739976
(Epoch 7 / 10) train acc: 0.460000; val_acc: 0.171000
(Iteration 15 / 20) loss: 1.645320
(Iteration 16 / 20) loss: 1.358373
(Epoch 8 / 10) train acc: 0.500000; val_acc: 0.185000
(Iteration 17 / 20) loss: 1.715330
(Iteration 18 / 20) loss: 0.983532
(Epoch 9 / 10) train acc: 0.560000; val_acc: 0.189000
(Iteration 19 / 20) loss: 0.895671
(Iteration 20 / 20) loss: 1.064744
(Epoch 10 / 10) train acc: 0.610000; val_acc: 0.195000
```

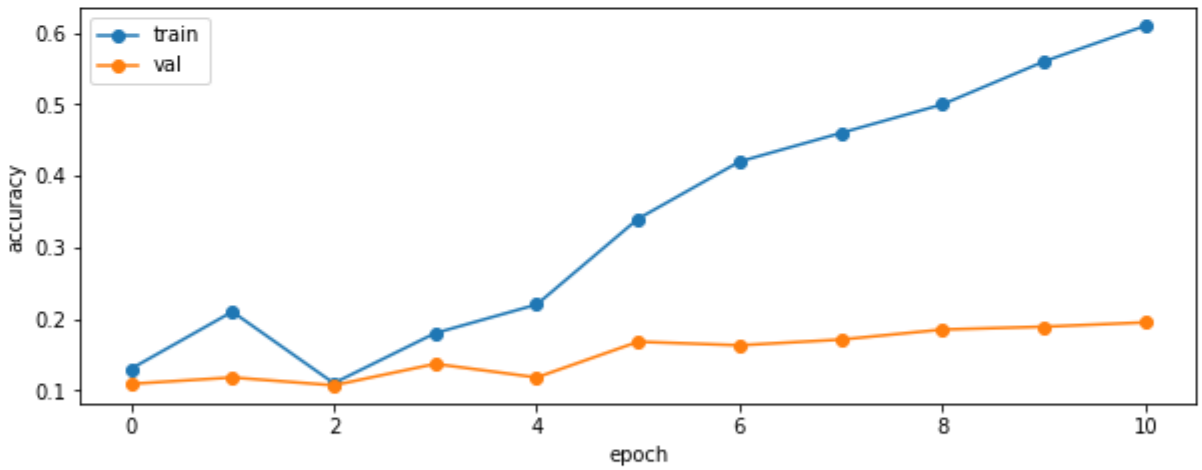
In [ ]:

```
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()
```





Train the network

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

In [ ]:

```
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: 2.304537
(Epoch 0 / 1) train acc: 0.095000; val_acc: 0.119000
(Iteration 21 / 980) loss: 2.276528
(Iteration 41 / 980) loss: 2.142111
(Iteration 61 / 980) loss: 2.283500
(Iteration 81 / 980) loss: 1.951992
(Iteration 101 / 980) loss: 1.672516
(Iteration 121 / 980) loss: 1.951228
(Iteration 141 / 980) loss: 2.007265
(Iteration 161 / 980) loss: 1.612091
(Iteration 181 / 980) loss: 1.968459
(Iteration 201 / 980) loss: 1.696876
(Iteration 221 / 980) loss: 1.822898
(Iteration 241 / 980) loss: 1.465408
(Iteration 261 / 980) loss: 1.741214
(Iteration 281 / 980) loss: 1.570055
(Iteration 301 / 980) loss: 1.998041
(Iteration 321 / 980) loss: 1.467284
(Iteration 341 / 980) loss: 1.398170
(Iteration 361 / 980) loss: 1.786516
(Iteration 381 / 980) loss: 1.405472
(Iteration 401 / 980) loss: 1.729057
(Iteration 421 / 980) loss: 1.572287
(Iteration 441 / 980) loss: 1.667767
(Iteration 461 / 980) loss: 1.575169
(Iteration 481 / 980) loss: 1.544825
(Iteration 501 / 980) loss: 1.476642
(Iteration 521 / 980) loss: 1.725957
(Iteration 541 / 980) loss: 1.408035
(Iteration 561 / 980) loss: 1.740912
(Iteration 581 / 980) loss: 1.508692
(Iteration 601 / 980) loss: 1.572061
(Iteration 621 / 980) loss: 1.210978
(Iteration 641 / 980) loss: 1.261266
(Iteration 661 / 980) loss: 1.410536
(Iteration 681 / 980) loss: 1.493152
(Iteration 701 / 980) loss: 1.848139
(Iteration 721 / 980) loss: 1.366808
(Iteration 741 / 980) loss: 1.375927
(Iteration 761 / 980) loss: 1.274215
(Iteration 781 / 980) loss: 1.417234
(Iteration 801 / 980) loss: 1.227579
(Iteration 821 / 980) loss: 1.464433
(Iteration 841 / 980) loss: 1.272653
(Iteration 861 / 980) loss: 1.436028
(Iteration 881 / 980) loss: 1.338456
(Iteration 901 / 980) loss: 1.491120
(Iteration 921 / 980) loss: 1.541794
(Iteration 941 / 980) loss: 1.467144
(Iteration 961 / 980) loss: 1.396216
(Epoch 1 / 1) train acc: 0.500000; val_acc: 0.510000
```

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.

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]

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.

In [ ]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #

model = BestCNN(weight_scale=0.001, hidden_dim=500, reg=0.005, filter_size=3, num_filte

solver = Solver(model, data,
                num_epochs=5, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=100)
solver.train()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 4900) loss: 2.303826
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.136000
(Iteration 101 / 4900) loss: 1.865671
(Iteration 201 / 4900) loss: 1.576891
(Iteration 301 / 4900) loss: 1.769245
(Iteration 401 / 4900) loss: 1.645652
(Iteration 501 / 4900) loss: 1.399039
(Iteration 601 / 4900) loss: 1.687713
(Iteration 701 / 4900) loss: 1.349862
(Iteration 801 / 4900) loss: 1.243319
(Iteration 901 / 4900) loss: 1.223658
(Epoch 1 / 5) train acc: 0.573000; val_acc: 0.575000
(Iteration 1001 / 4900) loss: 1.181311
(Iteration 1101 / 4900) loss: 1.197070
(Iteration 1201 / 4900) loss: 1.172661
(Iteration 1301 / 4900) loss: 1.104846
(Iteration 1401 / 4900) loss: 1.112608
(Iteration 1501 / 4900) loss: 1.308356
(Iteration 1601 / 4900) loss: 1.106202
(Iteration 1701 / 4900) loss: 0.999598
(Iteration 1801 / 4900) loss: 1.024301
(Iteration 1901 / 4900) loss: 1.082049
(Epoch 2 / 5) train acc: 0.611000; val_acc: 0.595000
(Iteration 2001 / 4900) loss: 1.251875
(Iteration 2101 / 4900) loss: 0.891963
(Iteration 2201 / 4900) loss: 1.020011
(Iteration 2301 / 4900) loss: 1.096611
(Iteration 2401 / 4900) loss: 0.949318
(Iteration 2501 / 4900) loss: 1.014718
(Iteration 2601 / 4900) loss: 0.920442
(Iteration 2701 / 4900) loss: 0.866242
(Iteration 2801 / 4900) loss: 0.931062
(Iteration 2901 / 4900) loss: 0.868347
(Epoch 3 / 5) train acc: 0.734000; val_acc: 0.684000
(Iteration 3001 / 4900) loss: 0.877152
(Iteration 3101 / 4900) loss: 0.921016
(Iteration 3201 / 4900) loss: 0.765574
(Iteration 3301 / 4900) loss: 0.870688
(Iteration 3401 / 4900) loss: 0.776872
(Iteration 3501 / 4900) loss: 0.795678
(Iteration 3601 / 4900) loss: 0.727459
(Iteration 3701 / 4900) loss: 0.783006
(Iteration 3801 / 4900) loss: 0.950273
(Iteration 3901 / 4900) loss: 0.652905
(Epoch 4 / 5) train acc: 0.774000; val_acc: 0.692000
(Iteration 4001 / 4900) loss: 0.967348
(Iteration 4101 / 4900) loss: 0.760686
(Iteration 4201 / 4900) loss: 0.693271
(Iteration 4301 / 4900) loss: 0.789836
(Iteration 4401 / 4900) loss: 0.919256
(Iteration 4501 / 4900) loss: 0.689625
(Iteration 4601 / 4900) loss: 0.935902
(Iteration 4701 / 4900) loss: 0.803994
(Iteration 4801 / 4900) loss: 0.712820
(Epoch 5 / 5) train acc: 0.781000; val_acc: 0.680000
```

In [ ]:

```
# cnn .py

import numpy as np

from nndl.layers import *
from nndl.conv_layers import *
from utils.fast_layers import *
from nndl.layer_utils import *
from nndl.conv_layer_utils import *

import pdb

class ThreeLayerConvNet(object):
  """
  A three-layer convolutional network with the following architecture:

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

  The network operates on minibatches of data that have shape (N, C,
H, W)
  consisting of N images, each with height H and width W and with C
input
  channels.
  """

  def __init__(self, input_dim=(3, 32, 32), num_filters=32,
filter_size=7,
               hidden_dim=100, num_classes=10, weight_scale=1e-3,
reg=0.0,
               dtype=np.float32, use_batchnorm=False):
    """
    Initialize a new network.

    Inputs:
    - input_dim: Tuple (C, H, W) giving size of input data
    - num_filters: Number of filters to use in the convolutional layer
    - filter_size: Size of filters to use in the convolutional layer
    - hidden_dim: Number of units to use in the fully-connected hidden
layer
    - num_classes: Number of scores to produce from the final affine
layer.
    - weight_scale: Scalar giving standard deviation for random
initialization
      of weights.
    - reg: Scalar giving L2 regularization strength
    - dtype: numpy datatype to use for computation.
    """
    self.use_batchnorm = use_batchnorm
    self.params = {}
    self.reg = reg
```

```
    self.dtype = dtype


    # ==================================================================
#
    # YOUR CODE HERE:
    #   Initialize the weights and biases of a three layer CNN. To
initialize:
    #     - the biases should be initialized to zeros.
    #     - the weights should be initialized to a matrix with entries
    #         drawn from a Gaussian distribution with zero mean and
    #         standard deviation given by weight_scale.
    # ==================================================================
#

    C, H, W = input_dim

    # cnn params
    pad = (filter_size - 1) / 2
    conv_stride = 1
    pool_size = 2
    pool_stride = 2
    # output sizes after convolution and pooling
    H_out_conv, W_out_conv = int(1 + (H - filter_size + 2*pad) /
conv_stride), int(1 + (W - filter_size + 2*pad) / conv_stride)
    H_out_pool, W_out_pool = int(1 + (H_out_conv - pool_size) /
pool_stride), int(1 + (H_out_conv - pool_size) / pool_stride)

    # W1 = conv weights
    self.params['W1'] = weight_scale * np.random.randn(num_filters, C,
filter_size, filter_size)
    self.params['b1'] = np.zeros(num_filters)

    max_pool_output_size = int(num_filters * H_out_pool * W_out_pool)
    self.params['W2'] = weight_scale *
np.random.randn(max_pool_output_size, hidden_dim)
    self.params['b2'] = np.zeros(hidden_dim)
    self.params['W3'] = weight_scale * np.random.randn(hidden_dim,
num_classes)
    self.params['b3'] = np.zeros(num_classes)

    # ==================================================================
#
    # END YOUR CODE HERE
    # ==================================================================
#

    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)
```

```
  def loss(self, X, y=None):
    """
    Evaluate loss and gradient for the three-layer convolutional
network.

    Input / output: Same API as TwoLayerNet in fc_net.py.
    """
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']

    # pass conv_param to the forward pass for the convolutional layer
    filter_size = W1.shape[2]
    conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}

    # pass pool_param to the forward pass for the max-pooling layer
    pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

    scores = None

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the forward pass of the three layer CNN.  Store the
output
    #   scores as the variable "scores".
    # ================================================================
#

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

    # get scores for first layer (conv + relu + pool)
    h1, cache1 = conv_relu_pool_forward(x=X, w=W1, b=b1,
conv_param=conv_param, pool_param=pool_param)
    # get scores for second layer (fc)
    h2, cache2 = affine_relu_forward(x=h1, w=W2, b=b2) # get scores
for output layer (fc)
    scores, cache3 = affine_forward(x=h2, w=W3, b=b3)

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    if y is None:
      return scores

    loss, grads = 0, {}
```

```
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the backward pass of the three layer CNN.  Store the
grads
    #   in the grads dictionary, exactly as before (i.e., the gradient
of
    #   self.params[k] will be grads[k]).  Store the loss as "loss",
and
    #   don't forget to add regularization on ALL weight matrices.
    # ================================================================
#

    loss, dl = softmax_loss(x=scores, y=y)
    loss += 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.sum(W2**2) +
0.5*self.reg*np.sum(W3**2)
    dout, dW3, db3 = affine_backward(dl, cache3) # now backprop,
starting from the last affine layer
    dW3 += self.reg * W3
    dout, dW2, db2 = affine_relu_backward(dout, cache2)
    dW2 += self.reg * W2
    dx, dW1, db1 = conv_relu_pool_backward(dout, cache1)
    dW1 += self.reg * W1
    # now store all the gradients in the gradient dictionary
    grads["W1"] = dW1
    grads["W2"] = dW2
    grads["W3"] = dW3
    grads["b1"] = db1
    grads["b2"] = db2
    grads["b3"] = db3

    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    return loss, grads


class BestCNN(object):
  def __init__(self, input_dim=(3, 32, 32), num_filters=32,
filter_size=7, hidden_dim=100, num_classes=10, weight_scale=1e-3,
reg=0.0, dtype=np.float32, use_batchnorm=False):
    """
    Initialize a new network.
    Inputs:
    - input_dim: Tuple (C, H, W) giving size of input data
    - num_filters: Number of filters to use in the convolutional layer
    - filter_size: Size of filters to use in the convolutional layer
```

```
    - hidden_dim: Number of units to use in the fully-connected hidden
layer
    - num_classes: Number of scores to produce from the final affine
layer.
    - weight_scale: Scalar giving standard deviation for random
initialization
    of weights.
    - reg: Scalar giving L2 regularization strength
    - dtype: numpy datatype to use for computation. """
    self.use_batchnorm = use_batchnorm
    self.params = {}
    self.reg = reg
    self.dtype = dtype
    # ================================================================
# # YOUR CODE HERE:
    # # # # # #

    # plan
    # {conv relu conv relu pool} x 2 -> affine -> relu -> affine ->
output
    # bn plan
    # {conv bn relu conv bn relu pool} x 2 -> affine -> bn -> relu ->
affine -> output, thus need 5 bns

    C, H, W = input_dim
    # hyperparams to use
    pad = (filter_size - 1) / 2
    conv_stride = 1
    pool_size = 2
    pool_stride = 2

    # init
    self.params["W1"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, C, filter_size, filter_size))
    self.params["b1"] = np.zeros(num_filters)
    self.params["W2"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b2"] = np.zeros(num_filters)
    self.params["W3"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b3"] = np.zeros(num_filters)
    self.params["W4"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b4"] = np.zeros(num_filters)
    self.params["W5"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters*8*8, hidden_dim)) ## 8 because this is the
H_out_pool
    self.params["b5"] = np.zeros(hidden_dim)

    # output layer is different
```

```
    self.params["W6"] = np.random.normal(loc=0, scale=weight_scale,
size=(hidden_dim, num_classes))
    self.params["b6"] = np.zeros(num_classes)

    if self.use_batchnorm:
      for i in range(1,5):
        self.params['gamma'+str(i)] = np.ones(num_filters)
        self.params['beta'+str(i)] = np.zeros(num_filters)

    self.params['gamma5'] = np.ones(hidden_dim)
    self.params['beta5'] = np.zeros(hidden_dim)

    self.bn_params = []
    if self.use_batchnorm:
        self.bn_params = [{'mode': 'train'} for i in np.arange(5)]

    # ===============================================================
# # END YOUR CODE HERE
    # ===============================================================
#
    for k, v in self.params.items(): self.params[k] = v.astype(dtype)

  def loss(self, X, y=None):
    # print("input sahpe", X.shape)
    mode = 'test' if y is None else 'train'

    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']
    W4, b4 = self.params['W4'], self.params['b4']
    W5, b5 = self.params['W5'], self.params['b5']
    W6, b6 = self.params['W6'], self.params['b6']

    if self.use_batchnorm:
      for bn_param in self.bn_params:
          bn_param['mode'] = mode


    # take care of conv
    filter_size = W1.shape[2]
    conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}
    # take care of pool
    pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
    scores = None



    if not self.use_batchnorm:
      # forward pass w/o bn
      h1, cache1 = conv_relu_forward(x=X, w=W1, b=b1,
```

```
conv_param=conv_param) # conv relu

      h2, cache2 = conv_relu_forward(x=h1, w=W2, b=b2,
conv_param=conv_param) # conv relu
      h3, cache3 = max_pool_forward_fast(x=h2, pool_param=pool_param)
# pool
      h4, cache4 = conv_relu_forward(x=h3, w=W3, b=b3,
conv_param=conv_param) # conv relu
      h5, cache5 = conv_relu_forward(x=h4, w=W4, b=b4,
conv_param=conv_param) # conv relu
      h6, cache6 = max_pool_forward_fast(x=h5, pool_param=pool_param)
# pool
      # FC
      h7, cache7 = affine_relu_forward(x=h6, w=W5, b=b5)  # affine
      scores, cache8 = affine_forward(x=h7, w=W6, b=b6) # affine -
output



    bn_cache=[]
    # forward pass w bn
    if self.use_batchnorm:
      h1, cache1 = conv_bn_relu_forward(x=X, w=W1, b=b1,
conv_param=conv_param, gamma=self.params['gamma1'],
beta=self.params['beta1'], bn_param=self.bn_params[0]) # conv relu
      h2, cache2 = conv_bn_relu_forward(x=h1, w=W2, b=b2,
conv_param=conv_param, gamma=self.params['gamma2'],
beta=self.params['beta2'], bn_param=self.bn_params[1]) # conv relu
      h3, cache3 = max_pool_forward_fast(x=h2, pool_param=pool_param)
# pool
      h4, cache4 = conv_bn_relu_forward(x=h3, w=W3, b=b3,
conv_param=conv_param, gamma=self.params['gamma3'],
beta=self.params['beta3'], bn_param=self.bn_params[2]) # conv relu
      h5, cache5 = conv_bn_relu_forward(x=h4, w=W4, b=b4,
conv_param=conv_param, gamma=self.params['gamma4'],
beta=self.params['beta4'], bn_param=self.bn_params[3]) # conv relu
      h6, cache6 = max_pool_forward_fast(x=h5, pool_param=pool_param)
# pool
      # FC
      h7, cache7 = affine_batchnorm_relu_forward(x=h6, w=W5, b=b5,
gamma=self.params['gamma5'], beta=self.params['beta5'],
bn_params=self.bn_params[4])  # affine
      scores, cache8 = affine_forward(x=h7, w=W6, b=b6) # affine -
output


    if y is None:
      return scores
    loss, grads = 0, {}
```

```
    loss, dl = softmax_loss(x=scores, y=y) # then regularize the loss
    loss += 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.sum(W2**2) +
0.5*self.reg*np.sum(W3**2) + 0.5*self.reg*np.sum(W4**2) +
0.5*self.reg*np.sum(W5**2) + 0.5*self.reg*np.sum(W6**2)

    if not self.use_batchnorm:
      dout, dW6, db6 = affine_backward(dl, cache8) # affine
      dW6 += self.reg * W6
      dout, dW5, db5 = affine_relu_backward(dout, cache7) # affine
relu
      dW5 += self.reg * W5

      dout = max_pool_backward_fast(dout, cache6) # pool
      dout, dW4, db4 = conv_relu_backward(dout, cache5)# conv relu
      dW4 += self.reg * W4
      dout, dW3, db3 = conv_relu_backward(dout, cache4) # conv relu
      dW3 += self.reg * W3
      dout = max_pool_backward_fast(dout, cache3) # pool
      dout, dW2, db2 = conv_relu_backward(dout, cache2) # conv relu
      dW2 += self.reg * W2
      dout, dW1, db1 = conv_relu_backward(dout, cache1) # conv relu
      dW1 += self.reg * W1

    else:
      dout, dW6, db6 = affine_backward(dl, cache8) # affine
      dW6 += self.reg * W6
      dout, dW5, db5, dgamma5, dbeta5 =
affine_batchnorm_relu_backward(dout, cache7) # affine relu
      dW5 += self.reg * W5
      grads['gamma5'] = dgamma5
      grads['beta5'] = dbeta5

      dout = max_pool_backward_fast(dout, cache6) # pool
      dout, dW4, db4, dgamma4, dbeta4 = conv_bn_relu_backward(dout,
cache5)# conv relu
      dW4 += self.reg * W4
      grads['gamma4'] = dgamma4
      grads['beta4'] = dbeta4

      dout, dW3, db3, dgamma3, dbeta3 = conv_bn_relu_backward(dout,
cache4) # conv relu
      grads['gamma3'] = dgamma3
      grads['beta3'] = dbeta3
      dW3 += self.reg * W3

      dout = max_pool_backward_fast(dout, cache3) # pool
      dout, dW2, db2, dgamma2, dbeta2 = conv_bn_relu_backward(dout,
cache2) # conv relu
```

```
        grads['gamma2'] = dgamma2
        grads['beta2'] = dbeta2
        dW2 += self.reg * W2
        dout, dW1, db1, dgamma1, dbeta1 = conv_bn_relu_backward(dout,
cache1) # conv relu
        grads['gamma1'] = dgamma1
        grads['beta1'] = dbeta1
        dW1 += self.reg * W1


      # storage of w's and b's
      grads["W1"] = dW1
      grads["W2"] = dW2
      grads["W3"] = dW3
      grads["W4"] = dW4
      grads["W5"] = dW5
      grads["W6"] = dW6
      grads["b1"] = db1
      grads["b2"] = db2
      grads["b3"] = db3
      grads["b4"] = db4
      grads["b5"] = db5
      grads["b6"] = db6



      return loss, grads
```