CNN-Layers

February 27, 2018

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).

```
In [2]: ## 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.

In [21]:
```
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:  2.21214764175e-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.

```
In [41]: 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:  1.34022883001e-09
dw error:  9.48376673607e-10
db error:  5.01980019289e-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.

```
In [25]: 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:  4.16666651573e-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.

```
In [45]: 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)[

         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:  3.27562037461e-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.

```
In [46]: 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: 4.543105s
```

```
Fast: 0.391086s
Speedup: 11.616651x
Difference:  1.0573070187e-10

Testing conv_backward_fast:
Naive: 4.678907s
Fast: 0.012681s
Speedup: 368.969674x
dx difference:  1.82358483877e-11
dw difference:  5.32902849011e-13
db difference:  1.21859166569e-15
```

In [47]:
```
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: 0.269806s
fast: 0.002743s
speedup: 98.361495x
difference:  0.0
```

```
Testing pool_backward_fast:
Naive: 1.182779s
speedup: 70.107332x
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:

```
In [48]: 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_p
         dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_p
         db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_p

         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:  1.26034668417e-08
dw error:  7.79603272278e-10
db error:  2.21209041822e-11
```

```
In [49]: 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:  1.07832263092e-09
dw error:  5.34504540966e-10
db error:  1.45616078729e-11
```

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 27, 2018

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).

```
In [3]: ## 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))))
```

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

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.

In [7]:
```
# 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:  [ 10.13017373  10.18982157  10.69982472]
  Stds:  [ 4.1897632   3.95243651  4.08917049]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [  8.88178420e-17   5.89805982e-18  -3.44169138e-16]
  Stds:  [ 0.99999972  0.99999968  0.9999997 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [ 6.  7.  8.]
  Stds:  [ 2.99999915  3.99999872  4.9999985 ]
```

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.

In [10]:
```
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:  9.03050733449e-08
dgamma error:  1.33696449753e-11
dbeta error:  3.27558202154e-12
```

In [ ]:

CNN

February 27, 2018

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.

```
In [20]: # 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_gradien
         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))))
```

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

```
In [21]: # 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.

```
In [47]: 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, grad
```

```
W1 max relative error: 0.00019674547724440783
W2 max relative error: 0.004744448317317754
W3 max relative error: 1.8620765650509693e-05
b1 max relative error: 2.843206495737342e-05
b2 max relative error: 4.7446582991418503e-07
b3 max relative error: 5.011805049117704e-07
```

1.1.1 Overfit small dataset

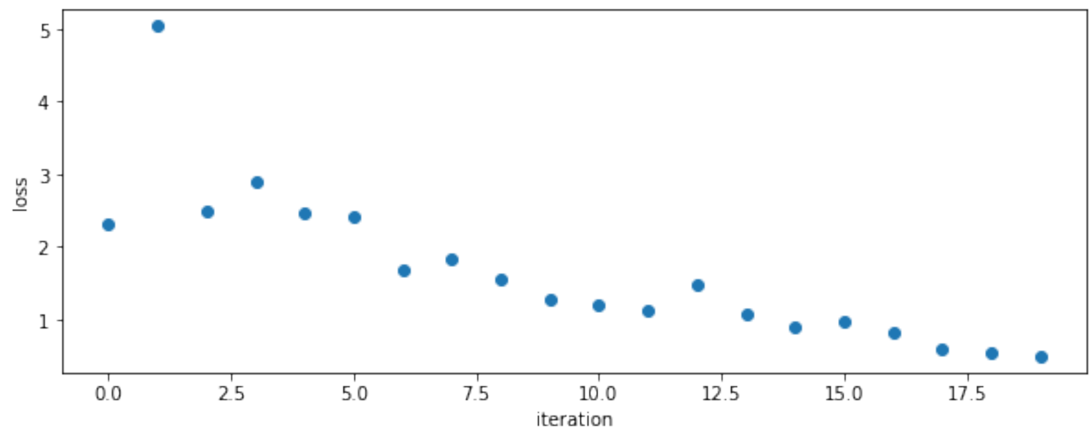
To check your CNN implementation, let's overfit a small dataset.

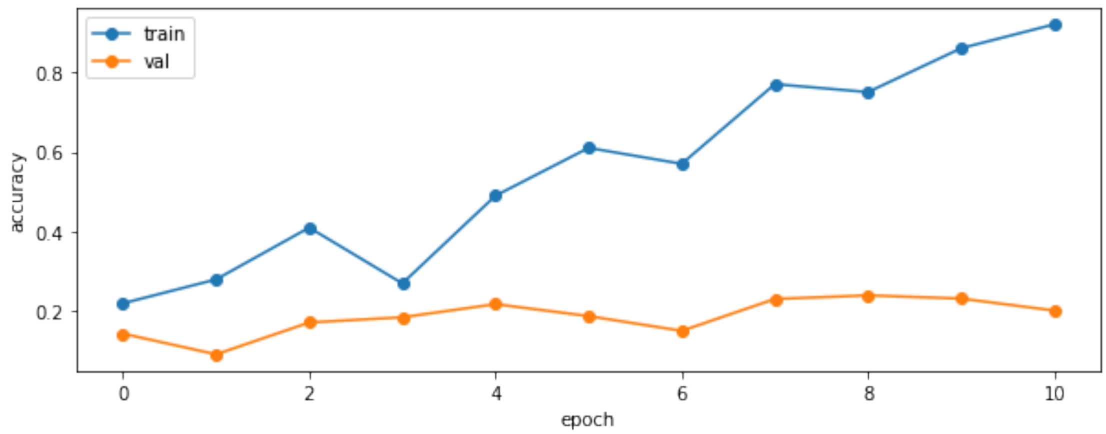
```
In [53]: num_train = 100
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': data['y_val'],
         }

         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: 2.320810
(Epoch 0 / 10) train acc: 0.220000; val_acc: 0.144000
(Iteration 2 / 20) loss: 5.053294
(Epoch 1 / 10) train acc: 0.280000; val_acc: 0.092000
(Iteration 3 / 20) loss: 2.484876
(Iteration 4 / 20) loss: 2.893186
```

```
(Epoch 2 / 10) train acc: 0.410000; val_acc: 0.172000
(Iteration 5 / 20) loss: 2.465218
(Iteration 6 / 20) loss: 2.405353
(Epoch 3 / 10) train acc: 0.270000; val_acc: 0.185000
(Iteration 7 / 20) loss: 1.683409
(Iteration 8 / 20) loss: 1.830656
(Epoch 4 / 10) train acc: 0.490000; val_acc: 0.218000
(Iteration 9 / 20) loss: 1.553348
(Iteration 10 / 20) loss: 1.272357
(Epoch 5 / 10) train acc: 0.610000; val_acc: 0.188000
(Iteration 11 / 20) loss: 1.196532
(Iteration 12 / 20) loss: 1.131260
(Epoch 6 / 10) train acc: 0.570000; val_acc: 0.151000
(Iteration 13 / 20) loss: 1.464487
(Iteration 14 / 20) loss: 1.074046
(Epoch 7 / 10) train acc: 0.770000; val_acc: 0.231000
(Iteration 15 / 20) loss: 0.881778
(Iteration 16 / 20) loss: 0.970694
(Epoch 8 / 10) train acc: 0.750000; val_acc: 0.240000
(Iteration 17 / 20) loss: 0.812045
(Iteration 18 / 20) loss: 0.596417
(Epoch 9 / 10) train acc: 0.860000; val_acc: 0.232000
(Iteration 19 / 20) loss: 0.545109
(Iteration 20 / 20) loss: 0.497141
(Epoch 10 / 10) train acc: 0.920000; val_acc: 0.202000
```

In [54]:
```
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.

```
In [55]: 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: 2.304772
(Epoch 0 / 1) train acc: 0.106000; val_acc: 0.105000
(Iteration 21 / 980) loss: 2.180104
(Iteration 41 / 980) loss: 2.025886
(Iteration 61 / 980) loss: 1.910054
```

```
(Iteration 81 / 980) loss: 2.039487
(Iteration 101 / 980) loss: 1.846337
(Iteration 121 / 980) loss: 1.544768
(Iteration 141 / 980) loss: 1.884846
(Iteration 161 / 980) loss: 1.796485
(Iteration 181 / 980) loss: 1.995384
(Iteration 201 / 980) loss: 1.834419
(Iteration 221 / 980) loss: 1.675321
(Iteration 241 / 980) loss: 1.658712
(Iteration 261 / 980) loss: 1.826970
(Iteration 281 / 980) loss: 1.469964
(Iteration 301 / 980) loss: 1.619737
(Iteration 321 / 980) loss: 1.546284
(Iteration 341 / 980) loss: 1.931794
(Iteration 361 / 980) loss: 1.700141
(Iteration 381 / 980) loss: 1.602551
(Iteration 401 / 980) loss: 1.728778
(Iteration 421 / 980) loss: 1.531434
(Iteration 441 / 980) loss: 2.094193
(Iteration 461 / 980) loss: 1.650501
(Iteration 481 / 980) loss: 1.591249
(Iteration 501 / 980) loss: 1.606730
(Iteration 521 / 980) loss: 1.779742
(Iteration 541 / 980) loss: 1.576076
(Iteration 561 / 980) loss: 1.642811
(Iteration 581 / 980) loss: 1.954226
(Iteration 601 / 980) loss: 2.204238
(Iteration 621 / 980) loss: 1.688987
(Iteration 641 / 980) loss: 1.315088
(Iteration 661 / 980) loss: 1.640437
(Iteration 681 / 980) loss: 1.296617
(Iteration 701 / 980) loss: 1.714731
(Iteration 721 / 980) loss: 1.422922
(Iteration 741 / 980) loss: 1.482807
(Iteration 761 / 980) loss: 1.459128
(Iteration 781 / 980) loss: 1.725105
(Iteration 801 / 980) loss: 1.607228
(Iteration 821 / 980) loss: 1.490936
(Iteration 841 / 980) loss: 1.380101
(Iteration 861 / 980) loss: 1.653268
(Iteration 881 / 980) loss: 1.398453
(Iteration 901 / 980) loss: 1.551622
(Iteration 921 / 980) loss: 1.545523
(Iteration 941 / 980) loss: 1.476686
(Iteration 961 / 980) loss: 1.657998
(Epoch 1 / 1) train acc: 0.492000; val_acc: 0.498000
```

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.

In [26]:
```
# ================================================================== #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================== #

model = ThreeLayerConvNet(num_filters=64, filter_size=3,
                          weight_scale=0.001, hidden_dim=500,
                          reg=0.001, use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=6, batch_size=200,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
#                 lr_decay = 0.9,
```

```python
                    verbose=True, print_every=20)
solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 1470) loss: 2.305346
(Epoch 0 / 6) train acc: 0.204000; val_acc: 0.189000
(Iteration 21 / 1470) loss: 1.773018
(Iteration 41 / 1470) loss: 1.572854
(Iteration 61 / 1470) loss: 1.567728
(Iteration 81 / 1470) loss: 1.503163
(Iteration 101 / 1470) loss: 1.515456
(Iteration 121 / 1470) loss: 1.564457
(Iteration 141 / 1470) loss: 1.342827
(Iteration 161 / 1470) loss: 1.397185
(Iteration 181 / 1470) loss: 1.440768
(Iteration 201 / 1470) loss: 1.514407
(Iteration 221 / 1470) loss: 1.338996
(Iteration 241 / 1470) loss: 1.228555
(Epoch 1 / 6) train acc: 0.612000; val_acc: 0.593000
(Iteration 261 / 1470) loss: 1.316691
(Iteration 281 / 1470) loss: 1.392568
(Iteration 301 / 1470) loss: 1.354490
(Iteration 321 / 1470) loss: 1.405908
(Iteration 341 / 1470) loss: 1.154451
(Iteration 361 / 1470) loss: 1.155922
(Iteration 381 / 1470) loss: 1.171903
(Iteration 401 / 1470) loss: 1.203219
(Iteration 421 / 1470) loss: 1.260604
(Iteration 441 / 1470) loss: 1.337409
(Iteration 461 / 1470) loss: 1.304432
(Iteration 481 / 1470) loss: 1.186095
(Epoch 2 / 6) train acc: 0.634000; val_acc: 0.619000
(Iteration 501 / 1470) loss: 1.124975
(Iteration 521 / 1470) loss: 1.271078
(Iteration 541 / 1470) loss: 1.217678
(Iteration 561 / 1470) loss: 1.207451
(Iteration 581 / 1470) loss: 1.086615
(Iteration 601 / 1470) loss: 1.217820
(Iteration 621 / 1470) loss: 1.186121
(Iteration 641 / 1470) loss: 1.107597
(Iteration 661 / 1470) loss: 1.180826
(Iteration 681 / 1470) loss: 1.181886
(Iteration 701 / 1470) loss: 1.325636
(Iteration 721 / 1470) loss: 1.179445
(Epoch 3 / 6) train acc: 0.700000; val_acc: 0.623000
(Iteration 741 / 1470) loss: 1.208147
```

```
(Iteration 761 / 1470) loss: 1.218818
(Iteration 781 / 1470) loss: 1.085747
(Iteration 801 / 1470) loss: 1.214951
(Iteration 821 / 1470) loss: 1.133458
(Iteration 841 / 1470) loss: 1.241980
(Iteration 861 / 1470) loss: 1.229727
(Iteration 881 / 1470) loss: 1.134834
(Iteration 901 / 1470) loss: 1.191719
(Iteration 921 / 1470) loss: 1.059599
(Iteration 941 / 1470) loss: 1.126035
(Iteration 961 / 1470) loss: 1.048853
(Epoch 4 / 6) train acc: 0.750000; val_acc: 0.637000
(Iteration 981 / 1470) loss: 0.973456
(Iteration 1001 / 1470) loss: 1.047568
(Iteration 1021 / 1470) loss: 1.211622
(Iteration 1041 / 1470) loss: 1.079860
(Iteration 1061 / 1470) loss: 1.028886
(Iteration 1081 / 1470) loss: 1.262685
(Iteration 1101 / 1470) loss: 1.110305
(Iteration 1121 / 1470) loss: 1.121391
(Iteration 1141 / 1470) loss: 1.040646
(Iteration 1161 / 1470) loss: 1.122613
(Iteration 1181 / 1470) loss: 1.177398
(Iteration 1201 / 1470) loss: 1.229780
(Iteration 1221 / 1470) loss: 1.093318
(Epoch 5 / 6) train acc: 0.743000; val_acc: 0.631000
(Iteration 1241 / 1470) loss: 1.169390
(Iteration 1261 / 1470) loss: 1.092743
(Iteration 1281 / 1470) loss: 1.109185
(Iteration 1301 / 1470) loss: 1.084981
(Iteration 1321 / 1470) loss: 1.042491
(Iteration 1341 / 1470) loss: 1.116905
(Iteration 1361 / 1470) loss: 1.021729
(Iteration 1381 / 1470) loss: 1.092293
(Iteration 1401 / 1470) loss: 0.917991
(Iteration 1421 / 1470) loss: 1.115744
(Iteration 1441 / 1470) loss: 1.044902
(Iteration 1461 / 1470) loss: 1.020942
(Epoch 6 / 6) train acc: 0.775000; val_acc: 0.656000
```

The following configuration yielded 65.6% validation accuracy: 64 filters 3x3 filters 0.001 weight scale 0.001 regularization 500 hidden dim Using batch normalization

For the solver: 6 epochs 200 batch size 1e-3 learning rate

```
In [ ]:
```

CNN.py

In [ ]:
```
import numpy as np

from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from nndl.layer_utils import *
from nndl.conv_layer_utils import *

import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

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 in
put
  channels.
  """

  def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_siz
e=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 l
ayer.
    - weight_scale: Scalar giving standard deviation for random initia
lization
      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 ini
tialize:
    #     - 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.
    # ================================================================
#
    self.bn_params = {}
    mu = 0
    stddev = weight_scale

    C = int(input_dim[0])
    H = int(input_dim[1])
    W = int(input_dim[2])

    #figure out padding
    pad = (filter_size -1)/2

    #default stride
    stride = 1

    #figure out filter dims
    H_f = (H + 2*pad - filter_size)/stride + 1
    W_f = (W + 2*pad - filter_size)/stride + 1

    self.params['W1'] = np.random.normal(mu, stddev, (num_filters, C,
filter_size, filter_size))
    self.params['b1'] = np.zeros(num_filters)
```

```
    pool_dim = 2
    pool_stride = 2
    H_pool = (H_f - pool_dim)/pool_stride + 1
    W_pool = (W_f - pool_dim)/pool_stride + 1

    self.params['W2'] = np.random.normal(mu, stddev, (int(num_filters*
H_pool*W_pool), hidden_dim))
    self.params['b2'] = np.zeros(hidden_dim)

    self.params['W3'] = np.random.normal(mu, stddev, (hidden_dim, num_
classes))
    self.params['b3'] = np.zeros(num_classes)

    #set up batchnorm
    if self.use_batchnorm is True:
        self.bn_params['bn_param1'] = {'mode': 'train', 'running_mean'
: np.zeros(num_filters), 'running_var': np.zeros(num_filters)}
        self.params['beta1'] = np.zeros(num_filters)
        self.params['gamma1'] = np.ones(num_filters)

        self.bn_params['bn_param2'] = {'mode': 'train', 'running_mean'
: np.zeros(hidden_dim), 'running_var': np.zeros(hidden_dim)}
        self.params['beta2'] = np.zeros(hidden_dim)
        self.params['gamma2'] = np.ones(hidden_dim)


    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
#     print(self.params.items())
    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 netwo
rk.

    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".
    # ================================================================
#
    #check batchnorm
    mode = 'test' if y is None else 'train'

    #set all to test if we want to test
    if self.use_batchnorm is True:
        for k, v in self.bn_params.items():
            v[mode] = mode

        bn_param1 = self.bn_params['bn_param1']
        bn_param2 = self.bn_params['bn_param2']

        beta1 = self.params['beta1']
        beta2 = self.params['beta2']

        gamma1 = self.params['gamma1']
        gamma2 = self.params['gamma2']



    #  conv - relu - 2x2 max pool - affine - relu - affine - softmax
    if self.use_batchnorm is True:
#        pizza()
        conv_out, conv_cache = conv_relu_pool_forward_batchnorm(X, W1,
b1, conv_param, pool_param, gamma1, beta1, bn_param1)
    else:
        #perform conv, relu, and pool
        conv_out, conv_cache = conv_relu_pool_forward(X, W1, b1, conv_
param, pool_param)

    #affine-relu layer
    N, F, H_out, W_out = conv_out.shape
    conv_out.reshape((N, F*H_out*W_out))
    if self.use_batchnorm is True:
        affine_out, affine_cache = affine_relu_forward_batchnorm(conv_
out, W2, b2, gamma2, beta2, bn_param2)
```

```
    else:
        affine_out, affine_cache = affine_relu_forward(conv_out, W2, b
2)

    #affine
    scores, affine2_cache = affine_forward(affine_out, W3, 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", a
nd
    #   don't forget to add regularization on ALL weight matrices.
    # ================================================================
#
    loss, grad_loss = softmax_loss(scores, y)

    #Add regularization to loss
    loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W
3**3))

    #affine_back -> relu_back -> affine_back -> conv_relu_pool_back
    #affine_backward returns dx, dw, db
    dx, grads['W3'], grads['b3'] = affine_backward(grad_loss, affine2_
cache)

    if self.use_batchnorm is True:
        dx, dw, db, dgamma2, dbeta2 = affine_relu_backward_batchnorm(d
x, affine_cache)
        grads['beta2'] = dbeta2
        grads['gamma2'] = dgamma2
    else:
        dx, dw, db = affine_relu_backward(dx, affine_cache)
    grads['W2'] = dw
    grads['b2'] = db
```

```
    #conv
    dx = np.reshape(dx, (N, F, H_out, W_out))
    if self.use_batchnorm is True:
        dx, dw, db, dgamma1, dbeta1 = conv_relu_pool_backward_batchnor
m(dx, conv_cache)
        grads['beta1'] = dbeta1
        grads['gamma1'] = dgamma1
    else:
        dx, dw, db = conv_relu_pool_backward(dx, conv_cache)
    grads['W1'] = dw
    grads['b1'] = db

    #regularization
    grads['W1'] += self.reg * W1
    grads['W2'] += self.reg * W2
    grads['W3'] += self.reg * W3




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

    return loss, grads


pass
```

In [ ]:
```
import numpy as np
from nndl.layers import *
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

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 filte
r 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 in
put.

  Returns a tuple of:
  - out: Output data, of shape (N, F, H', W') where H' and W' are give
n 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, C2, HH, WW = w.shape

  H_prime = 1 + (H + 2*pad - HH)/stride
  W_prime = 1 + (W+ 2*pad - WW)/stride

  #pad the input
  x_pad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad, pad)), mode='const
ant', constant_values=0)
  _, _, H_pad, W_pad = x_pad.shape

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

  #iterate through all data points
  for datapoint in range(N):
    x_pad_cur = x_pad[datapoint]

    h_loc, w_loc = -1, -1
    #go by height
    for hi in range(0, H_pad - HH + 1, stride):
      h_loc += 1
      #go by width
      for wi in range(0, W_pad - WW + 1, stride):
        w_loc += 1
        #first dim = : to get all channels
        x_all_channels = x_pad_cur[:, hi:hi+HH, wi:wi+WW]

        for filt in range(F):
          out[datapoint, filt, h_loc, w_loc] = np.sum(x_all_channels *
w[filt]) + b[filt]

      #reset height counter
      w_loc = -1


  # ================================================================ #
  # 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 laye
r.

  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='constan
t')
  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.
  # ================================================================ #

  #b is biases of shape (F,)
  #dout is N, F, out_height, out_width
  db = np.zeros((b.shape))
  for i in range(F):
    db[i] = np.sum(dout[:, i, :, :])


  #w: Filter weights of shape (F, C, HH, WW)
  F, C, HH, WW = w.shape
  dw = np.zeros((w.shape))
  for i in range(F):
    for j in range(C):
      for k in range(HH):
        for l in range(WW):
          #Input data of shape (N, C, H, W)
          derivative = dout[:, i, :, :] * xpad[:, j, k:k + out_height
* stride:stride, l:l + out_width * stride:stride]
```

```
          dw[i,j,k,l] = np.sum(derivative)


  #x: (N, C, H, W)
  _, _, H, W = x.shape
  #create dummy gradient -> will have same dimensions as x
  dx = np.zeros(x.shape)
  #pad the gradient dx
  dxpad = np.pad(dx, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='const
ant')
  H_prime = 1 + (H + 2*pad - HH)/stride
  W_prime = 1 + (W + 2*pad - WW)/stride

  for i in range(N): #for each data point
    for j in range(F):
      for k in range(0, int(H_prime)):#, stride):
        k_prime = k*stride
        for l in range(0, int(W_prime)):#, stride):
          l_prime = l*stride
          #multiply the weights of this filter by derivative dout
          derivative = w[j] * dout[i,j,k,l]
 #         print(dxpad.shape)
#          print(derivative.shape)
          dxpad[i, :, k_prime:k_prime + HH, l_prime:l_prime+WW] += der
ivative

  #extract derivative
  #dimensions need to be pulled out from H, W (, , H, W)
  dx = dxpad[:, :, pad:pad+H, pad:pad+W]
  # ================================================================ #
  # 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.
    # ================================================================== #
    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']
    stride = pool_param['stride']

    N, C, H, W = x.shape
    W_out = (W - pool_width)/stride + 1
    H_out = (H - pool_height)/stride + 1

    out = np.zeros((N, C, int(H_out), int(W_out)))

    for datapoint in range(N):
      #reduce by one dimension (datapoint #)
      x_cur = x[datapoint]

      h_loc, w_loc = -1, -1
      for hi in range(0, H-pool_height + 1, stride):
        h_loc += 1
        for wi in range(0, W-pool_width + 1, stride):
          w_loc += 1

          #this is the receptive field
          x_receptive_field = x_cur[:, hi:hi+pool_height, wi:wi+pool_wid
th]

          #iterate through all channels
          for c in range(C):
            out[datapoint, c, h_loc, w_loc] = np.max(x_receptive_field[c
])

        w_loc = -1





    # ================================================================== #
    # 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_pa
ram['pool_width'], pool_param['stride']

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

    dx = np.zeros(x.shape)
    N, C, H, W = x.shape
    H_prime = 1 + (H - pool_height)/stride
    W_prime = 1 + (W - pool_width)/stride
    for i in range(N):
      for j in range(C):
        for k in range(int(H_prime)):
          for l in range(int(W_prime)):
            k_prime = k * stride
            l_prime = l * stride

            #we want to only reward for the one we picked
            cur_window = x[i, j, k_prime:k_prime + pool_height, l_prime:
l_prime + pool_width]
            max_cur_window = np.max(cur_window)
            masked_window = (cur_window == max_cur_window)
            derivative = dout[i,j,k,l] * masked_window
            dx[i, j, k_prime:k_prime + pool_height, l_prime:l_prime + po
ol_width ] += derivative
    # ================================================================== #
    # 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, whil
e
        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 feature
s
      - running_var Array of shape (D,) giving running variance of featu
res

    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.
    # ================================================================ #
    #mode = bn_param['mode']
   # eps = bn_param.get['eps']
#  momentum = bn_param.get['momentum']
    N, C, H, W = x.shape

    #reshape the (N, C, H, W) array as an (N*H*W, C) array and perform b
atch normalization on this array.
    transpose = np.transpose(x, axes=(0,2,3,1))
    reshaped = transpose.reshape(N*H*W, C)
    bn_out, cache = batchnorm_forward(reshaped, gamma, beta, bn_param)

    #reshape again and swap
    out = bn_out.reshape(N, H, W, C).transpose(0, 3, 1, 2)



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

    return out, cache
```

```python
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, H, W = dout.shape
    transpose = np.transpose(dout, axes=(0,2,3,1))
    reshaped = transpose.reshape(N*H*W, C)
    dx_bn, dgamma_bn, dbeta_bn = batchnorm_backward(reshaped, cache)

    dx = dx_bn.reshape(N, H, W, C).transpose(0,3,1,2)
    dgamma = dgamma_bn
    dbeta = dbeta_bn

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

    return dx, dgamma, dbeta
```

layer_utils.py

In [ ]:
```
from nndl.layers import *
from cs231n.fast_layers import *
from nndl.conv_layers import *
"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""



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

  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_forward_batchnorm(x, w, b, gamma, beta, bn_param):
  a, fc_cache = affine_forward(x, w, b)
  a, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
  out, relu_cache = relu_forward(a)
  cache = (fc_cache, bn_cache, relu_cache)
  return out, cache

def affine_relu_backward(dout, cache):
  """
  Backward pass for the affine-relu convenience layer
  """
```

```
  fc_cache = cache[0]
  relu_cache = cache[1]
#  print("fc cache", fc_cache)
#  print(len(cache))

  da = relu_backward(dout, relu_cache)
  dx, dw, db = affine_backward(da, fc_cache)
  return dx, dw, db

def affine_relu_backward_batchnorm(dout, cache):
  fc_cache, bn_cache, relu_cache = cache
  da = relu_backward(dout, relu_cache)
  dbn, dgamma, dbeta = batchnorm_backward(da, bn_cache)
  dx, dw, db = affine_backward(dbn, fc_cache)

  return dx, dw, db, dgamma, dbeta

def affine_batchnorm_relu_forward(x, w, b, gamma, beta, bn_params):
  """
  Performs affine transformation, batchnorm, and ReLU

  Returns all caches

  BN forward takes: def batchnorm_forward(x, gamma, beta, bn_param):

  """
  out, forward_cache = affine_forward(x, w, b)
#  print("beta received: ", beta.shape)
  out, batchnorm_cache = batchnorm_forward(out, gamma, beta, bn_params
)
#  print("got dim: ", out.dim)
  out, relu_cache = relu_forward(out)

  total_cache = (forward_cache,relu_cache, batchnorm_cache)
#  print("returning out dim: ", out.shape)
  return out, total_cache


def affine_batchnorm_relu_backward(dout, cache):
  """
  Backward pass
  def batchnorm_backward(dout, cache):
  def relu_backward(dout, cache):


  """
  #unpack the cache tuple
  forward_cache, relu_cache, batchnorm_cache  = cache

  dx = relu_backward(dout, relu_cache)
```

```
  dx, dgamma, dbeta = batchnorm_backward(dx, batchnorm_cache)
  dx, dw, db = affine_backward(dx, forward_cache)

  gradients = dx, dw, db, dgamma, dbeta
  return gradients


"""
Functions for conv net without batchnorm
"""
def conv_relu_forward(x, w, b, conv_param):
#    conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}
  out, conv_cache = conv_forward_fast(x, w, b, conv_param)
  out, relu_cache = relu_forward(out)
  cache = (conv_cache, relu_cache)

  return out, cache

def conv_relu_backward(dout, cache):
  conv_cache, relu_cache = cache
  deriv = relu_backward(dout, relu_cache)
  dx, dw, db = conv_backward_fast(deriv, conv_cache)
  return dx, dw, db


#apply pooling
def conv_relu_pool_forward(x, w, b, conv_param, pool_param):
  out_conv_forward, conv_cache = conv_forward_fast(x, w, b, conv_param
)
  out_relu_forward, relu_cache = relu_forward(out_conv_forward)
  out, pool_cache = max_pool_forward_fast(out_relu_forward, pool_param
)

  cache = (conv_cache, relu_cache, pool_cache)
  return out, cache

def conv_relu_pool_backward(dout, cache):
  conv_cache, relu_cache, pool_cache = cache

  dpool = max_pool_backward_fast(dout, pool_cache)
  drelu = relu_backward(dpool, relu_cache)
  dx, dw, db = conv_backward_fast(drelu, conv_cache)

  return dx, dw, db

"""
Functions with batchnorm
"""
def conv_relu_forward_batchnorm(x, w, b, conv_param, gamma, beta, bn_p
aram):
```

```
  out, conv_cache = conv_forward_fast(x, w, b, conv_param)
  out, bn_cache = spatial_batchnorm_forward(out, gamma, beta, bn_param
)
  out, relu_cache = relu_forward(out)
  cache = (conv_cache, bn_cache, relu_cache)

  return out, cache

def conv_relu_backward_batchnorm(dout, cache):
  #relu back -> batchnorm back -> conv back
  conv_cache, bn_cache, relu_cache = cache

  deriv = relu_backward(dout, relu_cache)
  dbn, dgamma, dbeta = spatial_batchnorm_backward(deriv, bn_cache)
  dx, dw, db = conv_backward_fast(dbn, conv_cache)
  return dx, dw, db, dgamma, dbeta

def conv_relu_pool_forward_batchnorm(x, w, b, conv_param, pool_param,
gamma, beta, bn_param):
    #conv forward
    out_conv_forward, conv_cache = conv_forward_fast(x, w, b, conv_par
am)
    #batchnorm forward - def spatial_batchnorm_forward(x, gamma, beta,
bn_param):
    out_bn_forward, bn_cache = spatial_batchnorm_forward(out_conv_forw
ard, gamma, beta, bn_param)
    #relu forward
    out_relu_forward, relu_cache = relu_forward(out_bn_forward)
    #pool
#     print(pool_param, pool_param['pool_height'])
    out, pool_cache = max_pool_forward_fast(out_relu_forward, pool_par
am)

    cache = (conv_cache, bn_cache, relu_cache, pool_cache)
    return out, cache

def conv_relu_pool_backward_batchnorm(dout, cache):
  conv_cache, bn_cache, relu_cache, pool_cache = cache

  #pool -> relu -> batchnorm back -> conv
  dpool = max_pool_backward_fast(dout, pool_cache)
  drelu = relu_backward(dpool, relu_cache)

  dbn, dgamma, dbeta = spatial_batchnorm_backward(drelu, bn_cache)

  dx, dw, db = conv_backward_fast(dbn, conv_cache)

  grads = (dx, dw, db, dgamma, dbeta)
  return grads
```

layers.py

In [ ]:
```
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

def affine_forward(x, w, b):
  """
  Computes the forward pass for an affine (fully-connected) layer.

  The input x has shape (N, d_1, ..., d_k) and contains a minibatch of
N
  examples, where each example x[i] has shape (d_1, ..., d_k). We will
  reshape each input into a vector of dimension D = d_1 * ... * d_k, a
nd
  then transform it to an output vector of dimension M.

  Inputs:
  - x: A numpy array containing input data, of shape (N, d_1, ..., d_k
)
  - w: A numpy array of weights, of shape (D, M)
  - b: A numpy array of biases, of shape (M,)

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ================================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ================================================================ #

 # N = x.shape[0]
```

```
    #D = w.shape[0]
    #x_reshaped = np.reshape(x, (N,D))
    x_shape = x.shape
    #Reshaping it as N*D
    #x_shape[0] is equal to N
    x = x.reshape( [ x_shape[0], np.prod( x_shape[1:]) ] )
    out = x.dot(w) + b

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

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


def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b, x_shape = cache
    dx, dw, db = None, None, None

    # ================================================================== #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ================================================================== #

    #reshape x matrix to be N, D and multiply upstream for the chain rul
e
    """
    N = x.shape[0]
    D = w.shape[0]
    reshaped_x = np.reshape(x, (N, D))
    dw = reshaped_x.T.dot(dout)

    #derivative wrt x
    dx_raw = dout.dot(w.T)
    dx = np.reshape(dx_raw, x.shape)
```

```
  #sum derivative for bias
  db = np.sum(dout, axis=0)
  """
#  print
  dx = np.zeros_like(x)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)

  dx += dout.dot(w.T)
  dw += x.T.dot(dout)
  db += dout.sum( axis = 0)

  # Reshaping dx
  dx = dx.reshape(x_shape)

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

  return dx, dw, db

def relu_forward(x):
  """
  Computes the forward pass for a layer of rectified linear units (ReL
Us).

  Input:
  - x: Inputs, of any shape

  Returns a tuple of:
  - out: Output, of the same shape as x
  - cache: x
  """
  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU forward pass.
  # ================================================================ #

#  out = np.maximum(0, x)
  out = np.maximum(x, np.zeros_like(x))


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

  cache = x
  return out, cache
```

```
def relu_backward(dout, cache):
  """
  Computes the backward pass for a layer of rectified linear units (Re
LUs).

  Input:
  - dout: Upstream derivatives, of any shape
  - cache: Input x, of same shape as dout

  Returns:
  - dx: Gradient with respect to x
  """
  x = cache

  # ================================================================== #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ================================================================== #

  #ReLU backward pass multiplies the dout by the indicator function
  #arr[arr > 255] = x
  dx = dout

  #apply indicator. Uses < and not <= because 0 is undefined for ReLU
  dx[x < 0] = 0
  # ================================================================== #
  # END YOUR CODE HERE
  # ================================================================== #

  return dx

def batchnorm_forward(x, gamma, beta, bn_param):
  """
  Forward pass for batch normalization.

  During training the sample mean and (uncorrected) sample variance ar
e
  computed from minibatch statistics and used to normalize the incomin
g data.
  During training we also keep an exponentially decaying running mean
of the mean
  and variance of each feature, and these averages are used to normali
ze data
  at test-time.

  At each timestep we update the running averages for mean and varianc
e using
  an exponential decay based on the momentum parameter:

  running_mean = momentum * running_mean + (1 - momentum) * sample_mea
```

```
n
  running_var = momentum * running_var + (1 - momentum) * sample_var

  Note that the batch normalization paper suggests a different test-ti
me
  behavior: they compute sample mean and variance for each feature usi
ng a
  large number of training images rather than using a running average.
For
  this implementation we have chosen to use running averages instead s
ince
  they do not require an additional estimation step; the torch7 implem
entation
  of batch normalization also uses running averages.

  Input:
  - x: Data of shape (N, D)
  - gamma: Scale parameter of shape (D,)
  - beta: Shift paremeter of shape (D,)
  - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance.
    - running_mean: Array of shape (D,) giving running mean of feature
s
    - running_var Array of shape (D,) giving running variance of featu
res

  Returns a tuple of:
  - out: of shape (N, D)
  - cache: A tuple of values needed in the backward pass
  """
  mode = bn_param['mode']
  eps = bn_param.get('eps', 1e-5)
  momentum = bn_param.get('momentum', 0.9)


#  print("received x: ", x.shape)
  N, D = x.shape
  running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtyp
e))
  running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype)
)

  out, cache = None, None
  if mode == 'train':

    # ==================================================================
#
    # YOUR CODE HERE:
```

```
    #   A few steps here:
    #     (1) Calculate the running mean and variance of the minibatch
.
    #     (2) Normalize the activations with the batch mean and varian
ce.
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass i
n
    #         the 'cache' variable.
    # ================================================================
#
    sample_mean = np.mean(x, axis=0)
    sample_var = np.var(x, axis=0)

    running_mean = momentum*running_mean + (1-momentum)*sample_mean
    running_var = momentum*running_var + (1-momentum)*sample_var

    x_hat = (x - sample_mean) / np.sqrt(sample_var + eps)
    out = x_hat*gamma + beta

    #store in cache
    cache = (mode, x, gamma, sample_mean, sample_var, x_hat, out, eps)

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

  elif mode == 'test':

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the testing time normalized activations.  Normalize
using
    #   the running mean and variance, and then scale and shift approp
riately.
    #   Store the output as 'out'.
    # ================================================================
#

    stddev = np.sqrt(running_var + eps)
    x_hat = (x - running_mean)/stddev
    out = x_hat*gamma + beta

    #store in cache
    cache = (mode, x, gamma, x_hat, out, eps, stddev)
```

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

  else:
    raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

  # Store the updated running means back into bn_param
  bn_param['running_mean'] = running_mean
  bn_param['running_var'] = running_var

#  print(out.shape)
  return out, cache

def batchnorm_backward(dout, cache):
  """
  Backward pass for batch normalization.

  For this implementation, you should write out a computation graph fo
r
  batch normalization on paper and propagate gradients backward throug
h
  intermediate nodes.

  Inputs:
  - dout: Upstream derivatives, of shape (N, D)
  - cache: Variable of intermediates from batchnorm_forward.

  Returns a tuple of:
  - dx: Gradient with respect to inputs x, of shape (N, D)
  - dgamma: Gradient with respect to scale parameter gamma, of shape (
D,)
  - dbeta: Gradient with respect to shift parameter beta, of shape (D,
)
  """
  dx, dgamma, dbeta = None, None, None
  mode = cache[0]
  # ================================================================== #
  # YOUR CODE HERE:
  #   Implement the batchnorm backward pass, calculating dx, dgamma, a
nd dbeta.
  # ================================================================== #
  if(mode == 'train'):
    mode, x, gamma, sample_mean, sample_var, x_hat, out, eps = cache
#    print(cache)

    N, D = x.shape
```

```
    dl_dbeta = np.sum(dout, axis=0)
#    print(dout.shape, x_hat.shape)
    dl_dgamma = np.sum(dout*x_hat, axis=0)
    dl_dx = dout*gamma


    dl_da = (1/np.sqrt(sample_var + eps))*dl_dx
    dl_du = -(1/np.sqrt(sample_var+eps))*np.sum(dl_dx, axis=0)

    dl_de = -0.5*(1/(sample_var+eps))*(x_hat)*dl_dx

    dl_dvar = np.sum(dl_de, axis=0)

    dl_da = (1/(np.sqrt(sample_var + eps)))*dl_dx

    dx = dl_da + 2*((x-sample_mean)/N)*dl_dvar + (1/N)*dl_du
    dgamma = dl_dgamma
    dbeta = dl_dbeta

  elif(mode == 'test'):
    mode, x, gamma, x_hat, out, eps, stddev = cache
    dl_dbeta = np.sum(dout, axis=0)
    dl_dgamma = np.sum(dout*x_hat, axis=0)
    dx = (gamma*dout)/stddev

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

  return dx, dgamma, dbeta

def dropout_forward(x, dropout_param):
  """
  Performs the forward pass for (inverted) dropout.

  Inputs:
  - x: Input data, of any shape
  - dropout_param: A dictionary with the following keys:
    - p: Dropout parameter. We drop each neuron output with probabilit
y p.
    - mode: 'test' or 'train'. If the mode is train, then perform drop
out;
      if the mode is test, then just return the input.
    - seed: Seed for the random number generator. Passing seed makes t
his
      function deterministic, which is needed for gradient checking bu
t not in
      real networks.

  Outputs:
```

```
  - out: Array of the same shape as x.
  - cache: A tuple (dropout_param, mask). In training mode, mask is th
e dropout
    mask that was used to multiply the input; in test mode, mask is No
ne.
  """
  p, mode = dropout_param['p'], dropout_param['mode']
  if 'seed' in dropout_param:
    np.random.seed(dropout_param['seed'])

  mask = None
  out = None

  if mode == 'train':
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during training ti
me.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ================================================================
#
#    print(x.shape)
    mask = (np.random.rand(*x.shape) < (1-p)) / (1-p)
    out = x * mask
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

  elif mode == 'test':

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during test time.
    # ================================================================
#

    out = x
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

  cache = (dropout_param, mask)
  out = out.astype(x.dtype, copy=False)
```

```
  return out, cache

def dropout_backward(dout, cache):
  """
  Perform the backward pass for (inverted) dropout.

  Inputs:
  - dout: Upstream derivatives, of any shape
  - cache: (dropout_param, mask) from dropout_forward.
  """
  dropout_param, mask = cache
  mode = dropout_param['mode']

  dx = None
  if mode == 'train':
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during training t
ime.
    # ================================================================
#
    dx = dout*mask
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
  elif mode == 'test':
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during test time.
    # ================================================================
#
    dx = dout
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
  return dx

def svm_loss(x, y):
  """
  Computes the loss and gradient using for multiclass SVM classificati
on.

  Inputs:
```

```
  - x: Input data, of shape (N, C) where x[i, j] is the score for the
jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i
] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """
  N = x.shape[0]
  correct_class_scores = x[np.arange(N), y]
  margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.
0)
  margins[np.arange(N), y] = 0
  loss = np.sum(margins) / N
  num_pos = np.sum(margins > 0, axis=1)
  dx = np.zeros_like(x)
  dx[margins > 0] = 1
  dx[np.arange(N), y] -= num_pos
  dx /= N
  return loss, dx


def softmax_loss(x, y):
  """
  Computes the loss and gradient for softmax classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the
jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i
] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """
  eps = 1e-7
  probs = np.exp(x - np.max(x, axis=1, keepdims=True))
  probs /= np.sum(probs, axis=1, keepdims=True)
  N = x.shape[0]
  loss = -np.sum(np.log(probs[np.arange(N), y] +eps)) / N
  dx = probs.copy()
  dx[np.arange(N), y] -= 1
  dx /= N
  return loss, dx
```

optim.py

In [ ]:
```
import numpy as np

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use
in the
ECE 239AS class at UCLA.  This includes the descriptions of what code
to
implement as well as some slight potential changes in variable names t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

"""
This file implements various first-order update rules that are commonl
y used for
training neural networks. Each update rule accepts current weights and
the
gradient of the loss with respect to those weights and produces the ne
xt set of
weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:
  - w: A numpy array giving the current weights.
  - dw: A numpy array of the same shape as w giving the gradient of th
e
    loss with respect to w.
  - config: A dictionary containing hyperparameter values such as lear
ning rate,
    momentum, etc. If the update rule requires caching values over man
y
    iterations, then config will also hold these cached values.

Returns:
  - next_w: The next point after the update.
  - config: The config dictionary to be passed to the next iteration o
f the
    update rule.

NOTE: For most update rules, the default learning rate will probably n
ot perform
well; however the default values of the other hyperparameters should w
```

```
ork well
for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w
and
setting next_w equal to w.
"""


def sgd(w, dw, config=None):
  """
  Performs vanilla stochastic gradient descent.

  config format:
  - learning_rate: Scalar learning rate.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)

  w -= config['learning_rate'] * dw
  return w, config


def sgd_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to stor
e a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn'
t there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else
sets it to zero.

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weigh
ts
  #   as next_w, and store the updated velocity as v.
  # ================================================================ #
  v = config['momentum']*v - config['learning_rate']*dw
  next_w = w + v
```

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

  config['velocity'] = v

  return next_w, config

def sgd_nesterov_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with Nesterov momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to stor
e a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn'
t there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else
sets it to zero.

  # =============================================================== #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weigh
ts
  #   as next_w, and store the updated velocity as v.
  # =============================================================== #
  v_old = v
  v = config['momentum']*v - config['learning_rate']*dw
  next_w = w + v + config['momentum']*(v-v_old)

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

  config['velocity'] = v

  return next_w, config

def rmsprop(w, dw, config=None):
  """
  Uses the RMSProp update rule, which uses a moving average of squared
gradient
```

```
  values to set adaptive per-parameter learning rates.

  config format:
  - learning_rate: Scalar learning rate.
  - decay_rate: Scalar between 0 and 1 giving the decay rate for the s
quared
    gradient cache.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero
.
  - beta: Moving average of second moments of gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('decay_rate', 0.99)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('a', np.zeros_like(w))

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement RMSProp.  Store the next value of w as next_w.  You ne
ed
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, so they can be used for future gradients. Conc
retely,
  #   config['a'] corresponds to "a" in the lecture notes.
  # ================================================================ #

  #hadamard product is taken care of by np multiplication
  a = config['a']
  beta = config['decay_rate']

  config['a'] = beta*a + (1-beta)*np.multiply(dw, dw)

  #update gradient
  next_w = w - np.multiply(config['learning_rate']/(np.sqrt(config['a'
])+config['epsilon']), dw)

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

  return next_w, config


def adam(w, dw, config=None):
  """
  Uses the Adam update rule, which incorporates moving averages of bot
h the
```

```
  gradient and its square and a bias correction term.

  config format:
  - learning_rate: Scalar learning rate.
  - beta1: Decay rate for moving average of first moment of gradient.
  - beta2: Decay rate for moving average of second moment of gradient.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero
.
  - m: Moving average of gradient.
  - v: Moving average of squared gradient.
  - t: Iteration number.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-3)
  config.setdefault('beta1', 0.9)
  config.setdefault('beta2', 0.999)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('v', np.zeros_like(w))
  config.setdefault('a', np.zeros_like(w))
  config.setdefault('t', 0)

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement Adam.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, and in config['v'] the moving average of the
  #   first moments.  Finally, store in config['t'] the increasing tim
e.
  # ================================================================ #

  beta1 = config['beta1']
  beta2 = config['beta2']
  v = config['v']
  a = config['a']

  #time update
  config['t'] = config['t'] + 1
  t = config['t']

  #first moment update (momentum-like)
  config['v'] = beta1*v + np.multiply(1-beta1, dw)

  #second moment update (gradient normalization)
  config['a'] = beta2*a + (1-beta2)*np.multiply(dw, dw)

  #bias correction in moments
  v_bar = (1/(1-beta1**t))*config['v']
  a_bar = (1/(1-beta2**t))*config['a']
```

```
  #gradient
  next_w = w - np.multiply(config['learning_rate']/(np.sqrt(a_bar)+con
fig['epsilon']), v_bar)


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

  return next_w, config
```