ECE 247 HW5

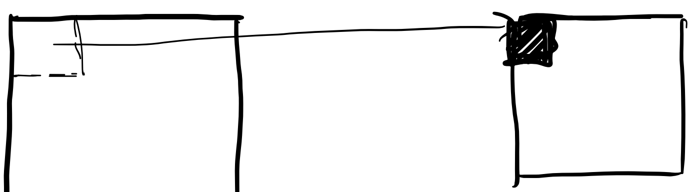
Yaman Yucel

Q1:

a) Receptive field of a neuron in CL_1 is $m_1 \times m_1$ or m_1.

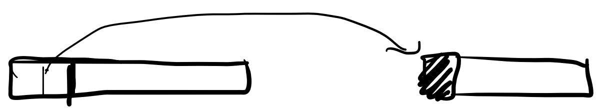
Suppose $m_1 = 2$,

2D:



CL1

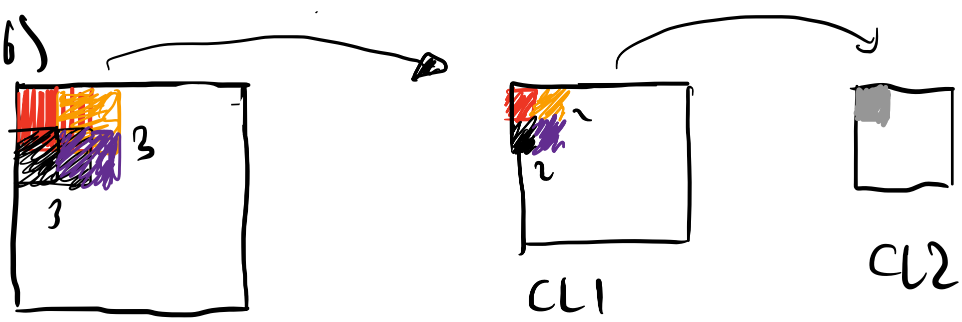
1D:



Each neuron in CL_1 is linked to $m_1 \times m_1$ patch in input. Therefore, RF is $m_1 \times m_1$

b)

2D

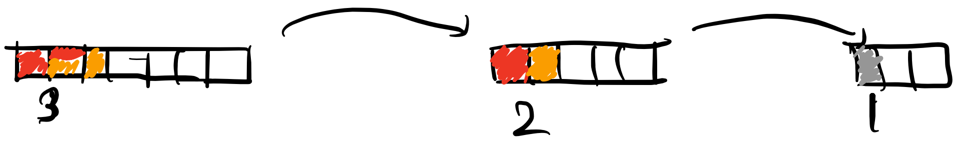


CL1

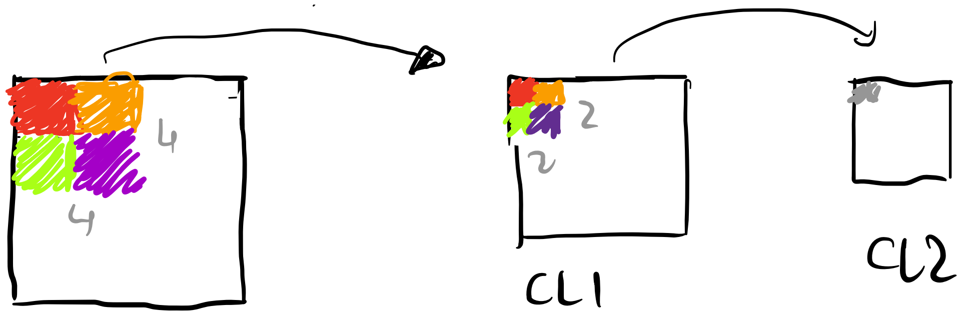
CL2

when $m_1 = m_2 = 2$, RF of a neuron in CL_2 is 3×3 or 3 which is $m_1 + m_2 - 1$ since we deduct the overlap. Therefore, receptive field of each neuron in CL_2 is $m_1 + m_2 - 1 \times m_1 + m_2 - 1$ or $m_1 + m_2 - 1$ when $stride_1 = stride_2 = 1$.

1D:



c)



It is not trivial to find receptive field, therefore one should consider a single layer.

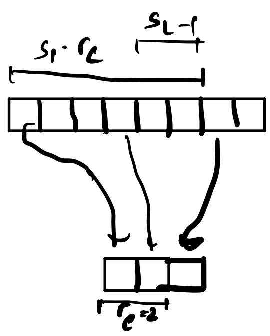
let r_ℓ denote the number of features in feature map CL_ℓ which contribute generate one feature in CL_ℓ. L is the last featuremap. $r_L = 1$

k_ℓ : kernel size (M_ℓ in our case)

s_ℓ : stride

$r_{L-1} = k_L$: we found that in a part.

Suppose we know, r_ℓ and we want to compute $r_{\ell-1}$.



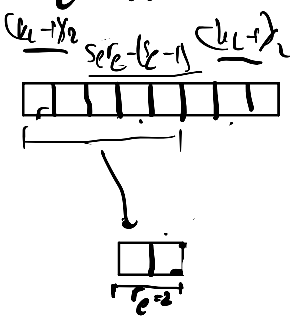
$k_\ell = 1$

$s_\ell = 3$

$s_\ell r_\ell$ will cover all features that contribute, however, it will be covered $s_\ell - 1$ more.
Therefore, formula will be

$$r_{\ell-1} = s_\ell r_\ell - (s_\ell - 1)$$

Suppose $k_\ell > 1$,



$k_\ell : 5$

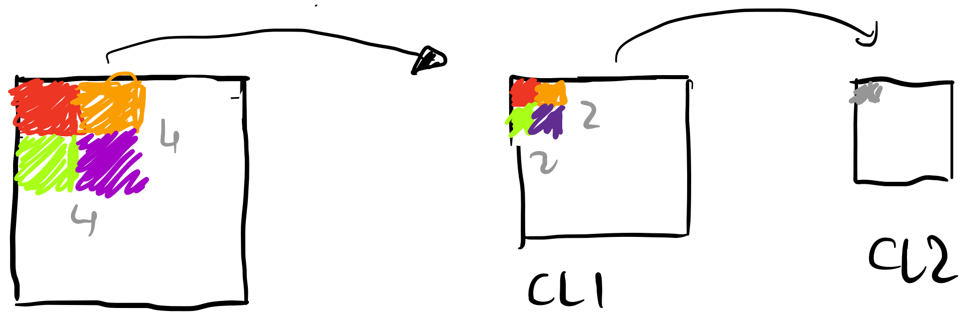
$s_\ell : 3$

Therefore, we will add $k_\ell - 1$ features to cover all.

$$r_{\ell-1} = s_\ell r_\ell - (s_\ell - 1) + k_\ell - 1$$
$$= s_\ell r_\ell + (k_\ell - s_\ell)$$

or in our case

$$= s_\ell r_\ell + (m_\ell - s_\ell)$$



CL1 CL2

$r_2 = 1$

$s_1 = 2$ $m_1 = 2$

$m_2 = 2$ $s_2 = ? = x$

$r_1 = s_2 \cancel{1} + m_2 - \cancel{s_2} = \boxed{m_2} \Rightarrow r_{L-1} = m_L = 2$ ✓

$r_0 = s_1 r_1 + m_1 - s_1 = 2 * 2 + 2 - 2 = 4$ ✓

Therefore, receptive field of CL_2 is

$r_2 = 1$

$r_1 = m_2$

$r_0 = \boxed{s_1 m_2 + m_1 - s_1}$ or $\boxed{(s_1 m_2 + m_1 - s_1 \times s_1 m_2 + m_1 - s_1)}$

Same formula can be used for any layer, you just need to assign that layer to be L.

Therefore, receptive field of CL_1 is

$r_1 = r_L = 1$

$r_0 = r_{L-1} = m_e = \boxed{m_1}$ or $\boxed{(m_1 \times m_1)}$

Note that receptive field of CL_1 is independent of stride, therefore unchanged.

However, receptive field of CL_2 can change.

when $s_1 = 1 \Rightarrow RF_1 = m_2 + m_1 - 1 > 0$

when $s_1 = 2 \Rightarrow RF_2 \quad 2m_2 + m_1 - 2 > 0$

For minimum m_2, m_1 $\quad RF_1 = 1$

$RF_2 = 2m_2 + m_1 - 2 = 1$

For $m_2 = 2, m_1 = 1$ $\quad RF_1 = 2$

$RF_2 = 4 + 1 - 2 = 3 > 2$

For $m_2=1, m_1=2$ $\quad RF_1 = 1$

$$\begin{aligned} RF_1 &= 1 \\ &\parallel \\ RF_2 &= 1 \end{aligned}$$

In result, CL_2 receptive field gets bigger as s_1 grows.

d) The recurrence relation is

$$r_{\ell-1} = s_\ell m_\ell + m_\ell - s_\ell \qquad r_L = 1$$

$$r_{L-1} = m_L$$

which has the solution,

$$r_0 = \sum_{\ell=1}^{L} \left((m_\ell - 1) \prod_{i=1}^{\ell-1} s_i \right) + 1$$

This was solved in practice problems, so I have only written the answer.

Note that we are asked for the k^{th} layer. Therefore assign layer L as k^{th} layer. Solution becomes

$$r_0 = \sum_{\ell=1}^{k} \left((m_\ell - 1) \prod_{i=1}^{\ell-1} s_i \right) + 1$$

Solution makes sense when $s_i = 1$,

$$r_0 = m_k + m_{k-1} + \dots + m_1 - k + 1$$

when $k=2$

$r_0 = m_2 - m_1 - 1$ which I found in part b.

Therefore, receptive layer of k^{th} layer is

$$\left(\sum_{e=1}^{k} \left((m_e - 1) \prod_{i=1}^{e-1} s_i \right) + 1 \times \sum_{e=1}^{k} \left((m_e - 1) \prod_{i=1}^{e-1} s_i \right) + 1 \right)$$

or

$$\sum_{e=1}^{k} \left((m_e - 1) \prod_{i=1}^{e-1} s_i \right) + 1$$

e) a) Increasing filter size m_i

b) Adding more layers

c) Increasing stride of filters except stride of last layer.

$$r_0 = \sum_{e=1}^{k} \left((m_e - 1) \prod_{i=1}^{e-1} s_i \right) + 1$$

a) $(m_e - 1) \uparrow$

b) $k \uparrow$

c) $s_i \uparrow$

CNN-Layers

February 26, 2023

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.

[1]:
```
## 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))))
```

0.2 Implementing CNN layers

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

0.2.1 Convolutional forward pass

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

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

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

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                           [-0.18387192, -0.2109216 ]],
                          [[ 0.21027089,  0.21661097],
                           [ 0.22847626,  0.23004637]],
                          [[ 0.50813986,  0.54309974],
                           [ 0.64082444,  0.67101435]]],
                         [[[-0.98053589, -1.03143541],
                           [-1.19128892, -1.24695841]],
                          [[ 0.69108355,  0.66880383],
                           [ 0.59480972,  0.56776003]],
                          [[ 2.36270298,  2.36904306],
                           [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around 1e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

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

0.2.2 Convolutional backward pass

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

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

[3]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
```

```
conv_param = {'stride': 1, 'pad': 1}

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

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

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

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

```
Testing conv_backward_naive function
dx error:  1.916590455692754e-08
dw error:  7.4069185122532045e-09
db error:  1.8782924169840352e-11
```

0.2.3 Max pool forward pass

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

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

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

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
```

```
                                [[ 0.20842105,  0.22315789],
                                 [ 0.26736842,  0.28210526]],
                                [[ 0.32631579,  0.34105263],
                                 [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be around 1e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

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

0.2.4 Max pool backward pass

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

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

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

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

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

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

```
Testing max_pool_backward_naive function:
dx error:  3.275620431353226e-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 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.

[6]:
```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: 2.644079s
Fast: 0.007051s
Speedup: 375.006695x
Difference:  1.0760953764912739e-10

Testing conv_backward_fast:
```

```
Naive: 4.269495s
Fast: 0.007568s
Speedup: 564.177625x
dx difference:  1.0012070746726426e-10
dw difference:  1.8655646296229697e-12
db difference:  3.9094552906934445e-15
```

[7]:
```
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: 0.213794s
fast: 0.004436s
speedup: 48.192562x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.581855s
speedup: 72.587877x
dx difference:  0.0
```

0.4 Implementation of cascaded layers

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

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

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

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

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

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

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

```
Testing conv_relu_pool
dx error:  7.939299810561729e-09
dw error:  6.562297642758617e-09
db error:  3.100564107304263e-11
```

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

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

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

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

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

```
Testing conv_relu:
dx error:  4.004510301971303e-09
dw error:  1.5116437241970764e-09
db error:  2.7145990866722925e-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 26, 2023

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.

[1]:
```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))))
```

0.2 Spatial batch normalization forward pass

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

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

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

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

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
```

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

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 9.77859078  9.04656259 10.2300046 ]
  Stds:  [4.68708021 4.17886643 4.67689384]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 5.55111512e-18  6.78623824e-16 -4.44089210e-17]
  Stds:  [0.99999964 0.99999977 0.9999998 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999891 3.99999909 4.99999902]
```

0.3 Spatial batch normalization backward pass

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

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

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

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

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

```
dx error:  5.586411465744636e-08
dgamma error:  1.7706480782599564e-12
dbeta error:  3.2917466860558243e-12
```

```
[ ]:
```

CNN

February 26, 2023

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.

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

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

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

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

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

import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from 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))))
```

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

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

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

1.1 Three layer CNN

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

The architecture of this CNN will be:

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

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

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

[3]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
```

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

```
W1 max relative error: 0.00014266947341464817
W2 max relative error: 0.010526612646796649
W3 max relative error: 0.00028064858554244487
b1 max relative error: 7.116381199585126e-05
b2 max relative error: 6.593377672518949e-07
b3 max relative error: 1.4712289610079897e-09
```

1.1.1 Overfit small dataset

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

[4]:
```
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()
```

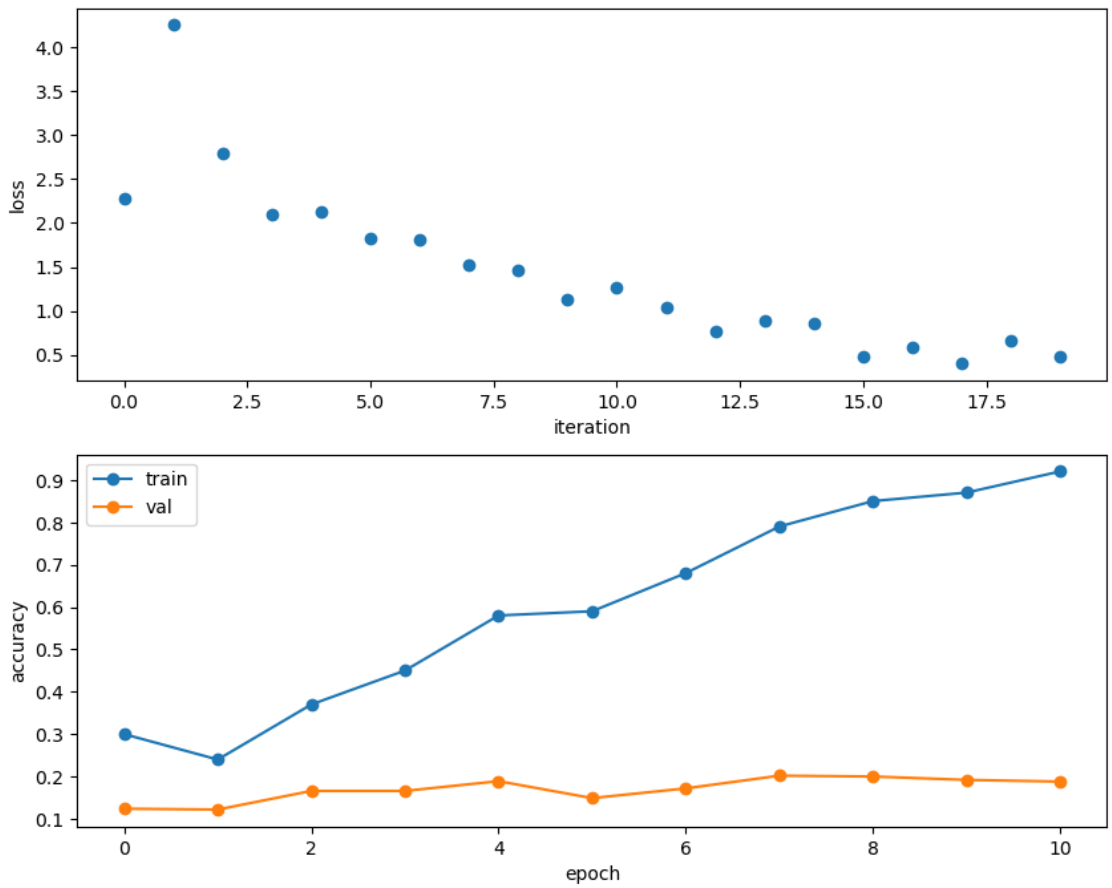
```
(Iteration 1 / 20) loss: 2.282205
(Epoch 0 / 10) train acc: 0.300000; val_acc: 0.124000
(Iteration 2 / 20) loss: 4.254238
(Epoch 1 / 10) train acc: 0.240000; val_acc: 0.122000
(Iteration 3 / 20) loss: 2.792451
(Iteration 4 / 20) loss: 2.095923
(Epoch 2 / 10) train acc: 0.370000; val_acc: 0.166000
(Iteration 5 / 20) loss: 2.132012
(Iteration 6 / 20) loss: 1.821241
```

```
(Epoch 3 / 10) train acc: 0.450000; val_acc: 0.166000
(Iteration 7 / 20) loss: 1.809644
(Iteration 8 / 20) loss: 1.516281
(Epoch 4 / 10) train acc: 0.580000; val_acc: 0.189000
(Iteration 9 / 20) loss: 1.468933
(Iteration 10 / 20) loss: 1.123248
(Epoch 5 / 10) train acc: 0.590000; val_acc: 0.149000
(Iteration 11 / 20) loss: 1.259533
(Iteration 12 / 20) loss: 1.045123
(Epoch 6 / 10) train acc: 0.680000; val_acc: 0.172000
(Iteration 13 / 20) loss: 0.767402
(Iteration 14 / 20) loss: 0.886044
(Epoch 7 / 10) train acc: 0.790000; val_acc: 0.202000
(Iteration 15 / 20) loss: 0.858364
(Iteration 16 / 20) loss: 0.478033
(Epoch 8 / 10) train acc: 0.850000; val_acc: 0.200000
(Iteration 17 / 20) loss: 0.577705
(Iteration 18 / 20) loss: 0.403593
(Epoch 9 / 10) train acc: 0.870000; val_acc: 0.192000
(Iteration 19 / 20) loss: 0.667685
(Iteration 20 / 20) loss: 0.481448
(Epoch 10 / 10) train acc: 0.920000; val_acc: 0.188000
```

[5]:

```
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.

[6]:
```
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: 2.304400
(Epoch 0 / 1) train acc: 0.139000; val_acc: 0.137000
(Iteration 21 / 980) loss: 2.674096
(Iteration 41 / 980) loss: 1.943157
(Iteration 61 / 980) loss: 2.355880
```

```
(Iteration 81 / 980) loss: 1.891819
(Iteration 101 / 980) loss: 2.012828
(Iteration 121 / 980) loss: 1.873694
(Iteration 141 / 980) loss: 1.755333
(Iteration 161 / 980) loss: 1.682914
(Iteration 181 / 980) loss: 1.815634
(Iteration 201 / 980) loss: 1.682235
(Iteration 221 / 980) loss: 2.011169
(Iteration 241 / 980) loss: 1.638945
(Iteration 261 / 980) loss: 1.787265
(Iteration 281 / 980) loss: 1.933533
(Iteration 301 / 980) loss: 1.681778
(Iteration 321 / 980) loss: 1.808253
(Iteration 341 / 980) loss: 1.513790
(Iteration 361 / 980) loss: 1.446696
(Iteration 381 / 980) loss: 1.577809
(Iteration 401 / 980) loss: 1.748466
(Iteration 421 / 980) loss: 1.770751
(Iteration 441 / 980) loss: 1.728824
(Iteration 461 / 980) loss: 1.434100
(Iteration 481 / 980) loss: 1.824666
(Iteration 501 / 980) loss: 1.649457
(Iteration 521 / 980) loss: 1.504636
(Iteration 541 / 980) loss: 1.610422
(Iteration 561 / 980) loss: 1.462938
(Iteration 581 / 980) loss: 1.568390
(Iteration 601 / 980) loss: 1.557461
(Iteration 621 / 980) loss: 2.082552
(Iteration 641 / 980) loss: 1.487003
(Iteration 661 / 980) loss: 1.418908
(Iteration 681 / 980) loss: 1.800945
(Iteration 701 / 980) loss: 1.792656
(Iteration 721 / 980) loss: 1.611114
(Iteration 741 / 980) loss: 1.502861
(Iteration 761 / 980) loss: 1.696393
(Iteration 781 / 980) loss: 1.589129
(Iteration 801 / 980) loss: 1.463187
(Iteration 821 / 980) loss: 1.648478
(Iteration 841 / 980) loss: 1.932970
(Iteration 861 / 980) loss: 1.610204
(Iteration 881 / 980) loss: 1.471085
(Iteration 901 / 980) loss: 1.505730
(Iteration 921 / 980) loss: 1.184835
(Iteration 941 / 980) loss: 1.605729
(Iteration 961 / 980) loss: 1.466829
(Epoch 1 / 1) train acc: 0.469000; val_acc: 0.488000
```

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

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

2.0.1 Things you should try:

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

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

2.0.2 Tips for training

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

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

[8]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #

model = ThreeLayerConvNet( input_dim=(3, 32, 32), num_filters=32, filter_size=7,
               hidden_dim=1000, num_classes=10, weight_scale=1e-3, reg=1e-3,
               dtype=np.float32, use_batchnorm=False)
solver = Solver(model, data,
                num_epochs=10, batch_size=1000,
                update_rule='adam',
                lr_decay = 0.99,
                optim_config={
                'learning_rate': 1e-3,
                },
                verbose=True, print_every=10)
solver.train()
```

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

```
(Iteration 1 / 490) loss: 2.306692
(Epoch 0 / 10) train acc: 0.119000; val_acc: 0.119000
(Iteration 11 / 490) loss: 2.125276
(Iteration 21 / 490) loss: 1.803713
(Iteration 31 / 490) loss: 1.692730
(Iteration 41 / 490) loss: 1.587318
(Epoch 1 / 10) train acc: 0.477000; val_acc: 0.511000
(Iteration 51 / 490) loss: 1.506039
(Iteration 61 / 490) loss: 1.455957
(Iteration 71 / 490) loss: 1.454727
(Iteration 81 / 490) loss: 1.356845
(Iteration 91 / 490) loss: 1.319754
(Epoch 2 / 10) train acc: 0.559000; val_acc: 0.560000
(Iteration 101 / 490) loss: 1.323770
(Iteration 111 / 490) loss: 1.252332
(Iteration 121 / 490) loss: 1.238275
(Iteration 131 / 490) loss: 1.236292
(Iteration 141 / 490) loss: 1.175816
(Epoch 3 / 10) train acc: 0.601000; val_acc: 0.584000
(Iteration 151 / 490) loss: 1.239669
(Iteration 161 / 490) loss: 1.101122
(Iteration 171 / 490) loss: 1.097917
(Iteration 181 / 490) loss: 1.050101
(Iteration 191 / 490) loss: 1.073609
(Epoch 4 / 10) train acc: 0.653000; val_acc: 0.604000
(Iteration 201 / 490) loss: 1.090483
(Iteration 211 / 490) loss: 1.060792
(Iteration 221 / 490) loss: 1.070918
(Iteration 231 / 490) loss: 0.957499
(Iteration 241 / 490) loss: 0.974188
(Epoch 5 / 10) train acc: 0.694000; val_acc: 0.653000
(Iteration 251 / 490) loss: 0.902507
(Iteration 261 / 490) loss: 0.865422
(Iteration 271 / 490) loss: 0.944982
(Iteration 281 / 490) loss: 0.884322
(Iteration 291 / 490) loss: 0.849965
(Epoch 6 / 10) train acc: 0.723000; val_acc: 0.650000
(Iteration 301 / 490) loss: 0.865890
(Iteration 311 / 490) loss: 0.842320
(Iteration 321 / 490) loss: 0.874008
(Iteration 331 / 490) loss: 0.832901
(Iteration 341 / 490) loss: 0.792152
(Epoch 7 / 10) train acc: 0.761000; val_acc: 0.630000
(Iteration 351 / 490) loss: 0.883844
```

```
(Iteration 361 / 490) loss: 0.781635
(Iteration 371 / 490) loss: 0.767589
(Iteration 381 / 490) loss: 0.802804
(Iteration 391 / 490) loss: 0.730505
(Epoch 8 / 10) train acc: 0.765000; val_acc: 0.653000
(Iteration 401 / 490) loss: 0.720166
(Iteration 411 / 490) loss: 0.714880
(Iteration 421 / 490) loss: 0.650941
(Iteration 431 / 490) loss: 0.713213
(Iteration 441 / 490) loss: 0.634855
(Epoch 9 / 10) train acc: 0.802000; val_acc: 0.686000
(Iteration 451 / 490) loss: 0.622079
(Iteration 461 / 490) loss: 0.646740
(Iteration 471 / 490) loss: 0.711556
(Iteration 481 / 490) loss: 0.690335
(Epoch 10 / 10) train acc: 0.823000; val_acc: 0.660000
```

[ ]:

```
"""
# I used here to see which parameters can be changed

model = ThreeLayerConvNet( input_dim=(3, 32, 32), num_filters=32, filter_size=7,
               hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=1e-3,
               dtype=np.float32, use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=10, batch_size=1000,
                update_rule='adam',
                lr_decay = 0.95,
                optim_config={
                'learning_rate': 1e-3,
                },
                verbose=True, print_every=10)
solver.train()
"""
```

```python
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
  H_out_shape = 1 + (H + 2 * pad - HH) // stride
  W_out_shape = 1 + (W + 2 * pad - WW) // stride

  out = np.zeros((N,F,H_out_shape,W_out_shape))

  x = np.pad(x, pad_width = ((0,0),(0,0),(pad,pad),(pad,pad)), mode = 'constant')
  for i in range(N):
    for j in range(F):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                x_selected = x[i,:,k * stride:(k*stride + HH), l * stride : (l * stride + WW)]
                w_selected = w[j,:,:,:]
                out[i,j,k,l] = np.sum(x_selected * w_selected) + b[j]

  # ================================================================ #
  # 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.
  # ================================================================ #
  N,F,H,W = x.shape

  H_out_shape = 1 + (H - f_height) // stride
  W_out_shape = 1 + (W - f_width) // stride

  dx = np.zeros(x.shape)
  dw = np.zeros(w.shape)
  db = np.zeros(b.shape)

  for i in range(N):
    for j in range(num_filts):
        if i == 0:
            db[j] += np.sum(dout[:,j,:,:])
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                dout_tmp = dout[i,j,k,l]
                dx[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)] += w[j,:,:,:] * dout_tmp
                dw[j,:,:,:] += x[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)] * dout_tmp
  dx = dx[:,:, pad:-pad, pad:-pad]
  # ================================================================ #
  # 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)
```

```python
  """
  out = None

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

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

  H_out_shape = 1 + (H - pool_height) // stride
  W_out_shape = 1 + (W - pool_width) // stride
  out = np.zeros((N,C, H_out_shape, W_out_shape))
  for i in range(N):
    for j in range(C):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
                out[i,j,k,l] = np.max(x_tmp)

  # ================================================================ #
  # 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
  H_out_shape = 1 + (H - pool_height) // stride
  W_out_shape = 1 + (W - pool_width) // stride

  dx = np.zeros((N,C,H,W))
  for i in range(N):
    for j in range(C):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
                dout_tmp = dout[i,j,k,l]
                din_mask = x_tmp == np.max(x_tmp)
                dx[i,j, k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)] += din_mask * dout_tmp

```

```
    # ================================================================ #
    # 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.
    # ================================================================ #
    N,C,H,W = x.shape
    x_flattened = (x.reshape((N,H,W,C))).reshape((N*W*H,C))
    out_bn, cache = batchnorm_forward(x_flattened, gamma,beta, bn_param = bn_param)
    out = (out_bn.reshape((N,W,H,C))).swapaxes(1,3)
    # ================================================================ #
    # 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.
    #
```

```python
    #   You may find it useful to use the batchnorm forward pass you
    #   implemented in HW #4.
    # ================================================================ #
    N,C,H,W = dout.shape
    dout_bn = dout.swapaxes(1,3).reshape((N*W*H,C))
    dx_bn, dgamma_bn, dbeta_bn = batchnorm_backward(dout_bn,cache)
    dx = dx_bn.reshape((N,C,H,W))
    dgamma = dgamma_bn.reshape((C,))
    dbeta = dbeta_bn.reshape((C,))
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx, dgamma, dbeta
```

```python
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
    shapes = {}
    shapes['W1'] = (num_filters, C, filter_size, filter_size)
    shapes['W2'] = ((H//2) *(W//2) * num_filters, hidden_dim)
    shapes['W3'] = (hidden_dim, num_classes)
    shapes['b1'] = num_filters
    shapes['b2'] = hidden_dim
    shapes['b3'] = num_classes

    for i in range(1,4):
        str_W = 'W' + str(i)
        str_b = 'b' + str(i)
        self.params[str_W] = np.random.normal(loc = 0.0, scale = weight_scale, size =
        shapes[str_W])
        self.params[str_b] = np.zeros(shapes[str_b])
```

```
    # ================================================================ #
    # 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".
    # ================================================================ #

    h1, cache1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
    h2, cache2 = affine_relu_forward(h1,W2,b2)
    scores, cache3 = affine_forward(h2,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", and
    #   don't forget to add regularization on ALL weight matrices.
    # ================================================================ #

    loss, dz = softmax_loss(scores,y)
    loss += 0.5*self.reg*(np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2))

    dh2,dw3, grads['b3'] = affine_backward(dz, cache3)
    dh1, dw2, grads['b2'] = affine_relu_backward(dh2, cache2)
    _, dw1, grads['b1'] = conv_relu_pool_backward(dh1, cache1)

    grads['W1'] = dw1 + self.reg * W1
    grads['W2'] = dw2 + self.reg * W2
    grads['W3'] = dw3 + self.reg * W3


    # ================================================================ #
```

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

    return loss, grads


pass

```

```python
import numpy as np
import pdb


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, and
    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)
    """
    out = None
    # ================================================================ #
    # 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.
    # ================================================================ #
    out = np.dot(x.reshape(x.shape[0],-1),w) + b


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

    cache = (x, w, b)
    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: 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:
    - 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 = cache
    dx, dw, db = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # Notice:
    #   dout is N x M
    #   dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M
    #   dw should be D x M; it relates to dout through multiplication with x, which is N
```

```
    x D after reshaping
    #   db should be M; it is just the sum over dout examples
    # ================================================================ #
    flattened_x = x.reshape(x.shape[0],-1)
    dx = np.dot(dout,w.T).reshape(x.shape)
    dw = np.dot(flattened_x.T,dout)
    db = np.sum(dout, axis = 0)

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

    return dx, dw, db

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

    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.
    # ================================================================ #
    relu = lambda x: x * (x > 0)
    out = relu(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 (ReLUs).

    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
    # ================================================================ #
    dx = dout * (x > 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 are
    computed from minibatch statistics and used to normalize the incoming 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 normalize data
    at test-time.

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

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

    Note that the batch normalization paper suggests a different test-time
    behavior: they compute sample mean and variance for each feature using a
    large number of training images rather than using a running average. For
    this implementation we have chosen to use running averages instead since
    they do not require an additional estimation step; the torch7 implementation
    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 features
      - running_var Array of shape (D,) giving running variance of features

    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)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    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 running mean and variance.
        #     (3) Scale and shift the normalized activations.  Store this
        #         as the variable 'out'
        #     (4) Store any variables you may need for the backward pass in
        #         the 'cache' variable.
        # ================================================================ #
        mean_x = np.mean(x,axis = 0)
        var_x = np.var(x, axis = 0)

        running_mean = momentum * running_mean + ( 1 - momentum ) * mean_x
        running_var = momentum * running_var + ( 1 - momentum ) * var_x

        standard_x = ( x - mean_x ) / ( np.sqrt(var_x + eps))

        out = gamma * standard_x + beta
        cache = (mean_x, var_x, standard_x, gamma, x, eps)

```

```python
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    elif mode == 'test':
        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the testing time normalized activation.  Normalize using
        #   the running mean and variance, and then scale and shift appropriately.
        #   Store the output as 'out'.
        # ================================================================ #

        standard_x = ( x - running_mean) / (np.sqrt(running_var))
        out = gamma * standard_x + beta
        #cache = []
        # ================================================================ #
        # 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

    return out, cache

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

    For this implementation, you should write out a computation graph for
    batch normalization on paper and propagate gradients backward through
    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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
    # ================================================================ #
    (mean_x, var_x, standard_x, gamma, x, eps) = cache
    sample_size = x.shape[0]
    sigma_x = np.sqrt(var_x + eps)

    dgamma = np.sum(standard_x * dout,axis = 0)
    dbeta = np.sum(dout, axis = 0)

    dL_dx_st = dout * gamma

    dx_st_da = 1 / sigma_x
    dL_da = dx_st_da * dL_dx_st
    da_dx = 1

    dx_st_de = -0.5 * ( dx_st_da ** 3) * (x - mean_x)
    dL_de = dx_st_de * dL_dx_st

    dL_dvar = np.sum(dL_de, axis = 0)
    dvar_dx = (2 * ( x - mean_x)) / sample_size
```

```

    dL_dmean = np.sum(-dL_da, axis = 0)
    dmean_dx = 1 / sample_size

    dx = da_dx * dL_da + dvar_dx * dL_dvar + dmean_dx * dL_dmean
    # ================================================================ #
    # 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 keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes this
        function deterministic, which is needed for gradient checking but not in
        real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
      mask that was used to multiply the input; in test mode, mask is None.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
    assert (0<p<=1), "Dropout probability is not in (0,1]"
    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 time.
        #   Store the masked and scaled activations in out, and store the
        #   dropout mask as the variable mask.
        # ================================================================ #

        mask = (np.random.rand(x.shape[0],x.shape[1]) < p) / 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)
```

```python

    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 time.
        # ================================================================ #

        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 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
    """
    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
    """

    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(np.maximum(probs[np.arange(N), y], 1e-8))) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

```python
import numpy as np

"""
This file implements various first-order update rules that are commonly 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 next 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 the
    loss with respect to w.
  - config: A dictionary containing hyperparameter values such as learning rate,
    momentum, etc. If the update rule requires caching values over many
    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 of the
    update rule.

NOTE: For most update rules, the default learning rate will probably not perform
well; however the default values of the other hyperparameters should work 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 store 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 weights
    #   as next_w, and the updated velocity as v.
    # ================================================================ #
```

```
    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = v + w
    # ================================================================ #
    # 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 store 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 weights
    #   as next_w, and the updated velocity as v.
    # ================================================================ #
    v_prev = v
    v = config['momentum'] * v_prev - config['learning_rate'] * dw
    next_w = v + w + config['momentum'] * (v - v_prev)
    # ================================================================ #
    # 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 squared
      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 need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, so they can be used for future gradients. Concretely,
```

```python
    #   config['a'] corresponds to "a" in the lecture notes.
    # ================================================================ #
    config['a'] = config['decay_rate'] * config['a'] + (1 - config['decay_rate']) * dw * dw
    c = 1 / (np.sqrt(config['a']) + config['epsilon'])
    next_w = w - config['learning_rate'] * c * 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 both 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 time.
    # ================================================================ #

    config['t'] += 1
    config['v'] = config['beta1'] * config['v'] + (1 - config['beta1']) * dw
    config['a'] = config['beta2'] * config['a'] + (1 - config['beta2']) * dw * dw

    v_tld = config['v'] / ( 1 - config['beta1'] ** config['t'])
    a_tld = config['a'] / ( 1 - config['beta2'] ** config['t'])

    c = 1 / (np.sqrt(a_tld) + config['epsilon'])
    next_w = w - config['learning_rate'] * v_tld * c

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

    return next_w, config
```