ConvolutionalNetworks

November 30, 2022

[5]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'hw5'
FOLDERNAME = "cs260/hw_5"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs260/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cs260/hw_5/cs260/datasets
/content/drive/My Drive/cs260/hw_5
```

1 Convolutional Networks

So far we have worked with deep fully connected networks, using them to explore different optimization strategies and network architectures. Fully connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

[7]:
```
# Setup cell.
import numpy as np
import matplotlib.pyplot as plt
from cs260.classifiers.cnn import *
from cs260.data_utils import get_CIFAR10_data
from cs260.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from cs260.layers import *
from cs260.fast_layers import *
from cs260.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))))
```

[17]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{k}: {v.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,)
```

2 Convolution: Naive Forward Pass

The core of a convolutional network is the convolution operation. In the file `cs260/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

[33]:
```
!pip install numpy --upgrade
from numpy.lib.stride_tricks import sliding_window_view

x = np.arange(4*4*3).reshape(3, 4, 4)
x = np.pad(x, ((0,), (1,), (1,)))

# N, C, H, W
filters_dim = (1, 1, 2, 2)
M3, K3, K1, K2 = filters_dim

inputs_dim = (1, 1, 4, 4)
N, N3, N1, N2 = inputs_dim



# H, W
stride = (1, 1)
S1, S2 = stride

n_every1 = append_after_every_index = S1 - 1
n_every2 = append_after_every_index = S2 - 1

pad = (0, 0)
P1, P2 = pad

# Shapes
M1 = (N1 - K1 + 2 * P1) // S1 + 1
M2 = (N2 - K2 + 2 * P2) // S2 + 1

transform_shape = (N, K3, K1, K2, 1, 1, M1, M2)
is1 = np.arange(1, M1)
is2 = np.arange(1, M2)

# Filters
filters = np.arange(np.prod(filters_dim)).reshape(filters_dim)
print('Original filters:\n', filters)

filters_reshaped = filters.reshape(M3, K1*K2*K3)
# print('\nReshaped filters:\n', filters_reshaped)

# Images
inputs = np.arange(np.prod(inputs_dim)).reshape(inputs_dim)
inputs = np.pad(inputs, pad_width=((0,), (0,), (P1,), (P2,)), mode='constant')
print(f'\nOriginal inputs {inputs.shape}:\n', inputs)
```

```
inputs_reshaped = sliding_window_view(inputs.T, window_shape=(K2, K1, K3, N)).
 ↪T#[..., ::S1, ::S2]#.reshape(N, K1*K2*K3, M1*M2)
#print(f'\nUnshaped inputs: {inputs_reshaped.shape}\n', inputs_reshaped)
inputs_reshaped = inputs_reshaped[..., ::S1, ::S2].reshape(N, K1*K2*K3, M1*M2)
# inputs_reshaped = sliding_window_view(inputs, window_shape=(K3, K1, K2),␣
 ↪axis=(1,2,3))[:, :, ::S1, ::S2]#.reshape((N, M1*M2, K1*K2*K3)).transpose((0,␣
 ↪2, 1))
#print(f'\nUntransposed inputs: {inputs_reshaped.shape}\n', inputs_reshaped)

# inputs_reshaped = inputs_reshaped.reshape((N, M1*M2, K1*K2*K3)).transpose((0,␣
 ↪2, 1))
# print(f'\nReshaped inputs: {inputs_reshaped.shape}\n', inputs_reshaped)

# reshaped_images = sliding_window_view(images, window_shape=(2, 2, 2),␣
 ↪axis=(1, 2, 3))[:, :, ::2, ::2].reshape((2, 4, 2*2*2))


# Activation maps
activation_maps = filters_reshaped @ inputs_reshaped
# activation_maps = np.tensordot(inputs_reshaped, filters_reshaped, axes=1)
# print('\nReshaped activation maps:\n', activation_maps)

activation_maps_reshaped = activation_maps.reshape((N, M3, M1, M2))
activation_maps_reshaped = np.ones_like(activation_maps_reshaped)
print(f'\nActivation maps {activation_maps_reshaped.shape}:\n',␣
 ↪activation_maps_reshaped)

# Derivatives
dx_reshaped = filters_reshaped.T @ activation_maps
# print('\ndx unshaped:\n', dx_reshaped)


dx_reshaped = np.expand_dims(dx_reshaped, axis=(0, 1, 2, 3)).
 ↪reshape(transform_shape)
# print('\ndx reshaped:\n', dx_reshaped)


if S1 > 1:
  dx_reshaped = np.insert(dx_reshaped, is1, [[0]]*n_every1, axis=6)

if S2 > 1:
  dx_reshaped = np.insert(dx_reshaped, is2, [[0]]*n_every2, axis=7)


dx_reshaped = dx_reshaped
```

```
#dx_reshaped = np.insert(dx_reshaped, i2, [[0]]*n_last2, axis=7)
#dx_reshaped = np.insert(dx_reshaped, is2, [[0]]*n_every2, axis=7)


#dx_reshaped = np.insert(dx_reshaped, (1, 2), 0, axis=7)
# print(f'\ndx unshaped: {dx_reshaped.shape}\n', dx_reshaped)


filter_new = np.rot90(filters, 2, axes=(2, 3))
# print(f'\nfilter flipped: {filter_new.shape}\n', filter_new)


filter_new_reshaped = filter_new.reshape(M3, K3*K1*K2)
filter_new_reshaped = np.concatenate(filter_new_reshaped.reshape(M3, K3, -1),␣
↪axis=1)
# print(f'\nfilter flipped reshaped: {filter_new_reshaped.shape}\n',␣
↪filter_new_reshaped)


activations_new = np.copy(activation_maps_reshaped)

if S1 > 1:
    activations_new = np.insert(activations_new, range(1, M1), [[0]]*n_every1,␣
↪axis=2)

if S2 > 1:
    activations_new = np.insert(activations_new, range(1, M2), [[0]]*n_every2,␣
↪axis=3)


# dout_row = np.concatenate(activations_new.reshape(N, M3, -1), axis=1)
dout_row = activations_new.reshape(N, 1, M3, -1)
#print(f'\ndout_row {dout_row .shape}:\n', dout_row)

x_col = sliding_window_view(inputs.T, window_shape=(activations_new.shape[3],␣
↪activations_new.shape[2], K3, N)).T.reshape((N, K3, activations_new.shape[3]␣
↪* activations_new.shape[2], -1))
#print(f'\nx_col {x_col.shape}:\n', x_col)

dw = np.moveaxis((dout_row @ x_col).sum(axis=0), 1, 0).reshape((M3, K3, K1, K2))
#print(f'\ndw {dw.shape}:\n', dw)


inputs_third = sliding_window_view(inputs.T, window_shape=(activations_new.
↪shape[3], activations_new.shape[2], K3, N)).T.reshape((N, K3,␣
↪activations_new.shape[3] * activations_new.shape[2], -1))
```

```
# print(f'\ndw {dw.shape}:\n', dw)

inputs_alt = sliding_window_view(inputs, window_shape=(N, K3, activations_new.
 ↪shape[2], activations_new.shape[3]))
# print(f'\nInputs reshaped {inputs_alt.shape}:\n', inputs_alt)
dw_alt = np.einsum('ijkl,mnopiqkl->jqop', activations_new, inputs_alt)
print(f'\ndw alt {dw_alt.shape}:\n', dw_alt)

activations_new = np.pad(activations_new, pad_width=((0,), (0,), (K1-1,),␣
 ↪(K2-1,)), mode='constant')
# print(f'\nactivations padded: {activations_new.shape}\n', activations_new)

M11 = (M1 - K1 + 2 * (K1-1)) + 1
M21 = (M2 - K2 + 2 * (K2-1)) + 1
# print(f'Expected area: {(M11, M21)}')

activations_reshaped = sliding_window_view(activations_new.T, window_shape=(K2,␣
 ↪K1, M3, N)).T.reshape(N, M3*K1*K2, -1)
# print(f'\nactivations reshaped: {activations_reshaped.shape}\n',␣
 ↪activations_reshaped)

activations_alt = sliding_window_view(activations_new, window_shape=(N, M3, K1,␣
 ↪K2))
# print(f'\nactivations alt: {activations_alt.shape}\n', activations_alt)

dx_reshaped = filter_new_reshaped @ activations_reshaped
dx = dx_reshaped.reshape((N, K3, N1, N2))
# print(f'\ndx: {dx.shape}\n', dx)

# M3, K3, K1, K2 | 1, 1, N1 N2 N M3 K1 K2
dx_alt = np.einsum('ijkl,mnopqikl->qjop', filter_new, activations_alt)
# print(f'\ndx alt: {dx_alt.shape}\n', dx_alt)
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(1.21.6)
Original filters:
 [[[[0 1]
   [2 3]]]]

Original inputs (1, 1, 4, 4):
 [[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]
   [12 13 14 15]]]]
```

```
Activation maps (1, 1, 3, 3):
 [[[[1 1 1]
   [1 1 1]
   [1 1 1]]]]

dw alt (1, 1, 2, 2):
 [[[[45 54]
   [81 90]]]]
```

[34]:
```
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 e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

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

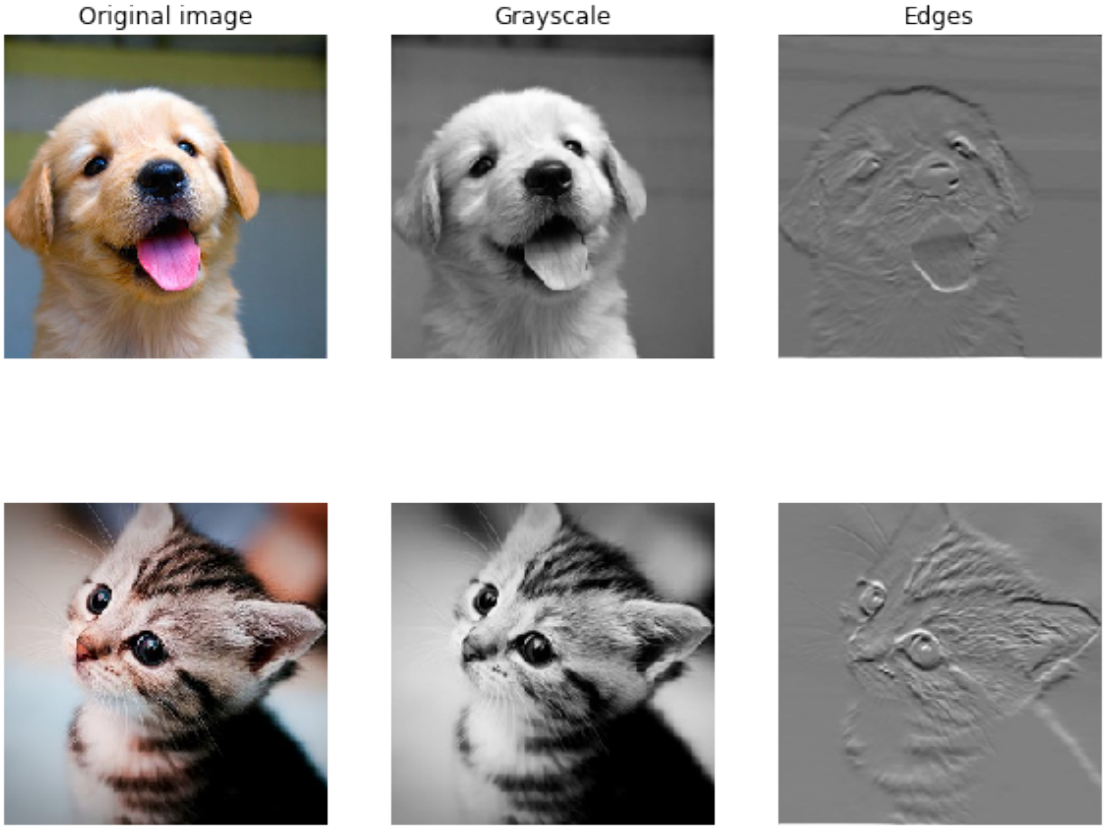
2.1 Aside: Image Processing via Convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

[35]:
```
from imageio import imread
from PIL import Image
```

```
kitten = imread('cs260/notebook_images/kitten.jpg')
puppy = imread('cs260/notebook_images/puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size)))
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_no_ax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
```

```
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()
```



3 Max-Pooling: Naive Forward Pass

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs260/layers.py`. Again, don't worry too much about

computational efficiency.

Check your implementation by running the following:

[36]:
```
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 on the order of e-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
```

4 Fast Layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs260/fast_layers.py`.

4.0.1 Execute the below cell, save the notebook, and restart the runtime

The fast convolution implementation depends on a Cython extension; to compile it, run the cell below. Next, save the Colab notebook (`File > Save`) and **restart the runtime** (`Runtime > Restart runtime`). You can then re-execute the preceeding cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

[37]:
```
# Remember to restart the runtime after executing this cell!
%cd /content/drive/My\ Drive/$FOLDERNAME/cs260/
!python setup.py build_ext --inplace
```

```
%cd /content/drive/My\ Drive/$FOLDERNAME/
```

```
/content/drive/My Drive/cs260/hw_5/cs260
running build_ext
/content/drive/My Drive/cs260/hw_5
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

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 following:

[8]:
```
# Rel errors should be around e-9 or less.
from cs260.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(260)
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: 16.609110s
Fast: 0.020297s
Speedup: 818.292037x
Difference:  1.679462625588412e-11

Testing conv_backward_fast:
Naive: 0.337749s
Fast: 0.012112s
Speedup: 27.886280x
dx difference:  1.385448899834027e-11
dw difference:  1.0
db difference:  9.102117060584485e-16
```

[9]:
```
# Relative errors should be close to 0.0.
from cs260.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(260)
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('fast: %fs' % (t2 - t1))
```

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

```
Testing pool_forward_fast:
Naive: 0.451049s
fast: 0.008313s
speedup: 54.257055x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.016980s
fast: 0.012728s
speedup: 1.334070x
dx difference:  0.0
```

5 Convolutional "Sandwich" Layers

In the previous assignment, we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs260/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check their usage.

[10]:
```
from cs260.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(260)
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)

# Relative errors should be around e-8 or less
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:  1.2963336641951403e-08
dw error:  1.21650111426998e-09
db error:  7.640313216365731e-11
```

[11]:
```
from cs260.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(260)
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)

# Relative errors should be around e-8 or less
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:  3.739879276480565e-09
dw error:  8.90579017628624e-09
db error:  3.875539243546206e-11
```

6 Three-Layer Convolutional Network

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs260/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

6.1 Sanity Check Loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization the loss should go up slightly.

[18]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print ('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print ('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.30258443722709
Initial loss (with regularization):  2.5091526634453136
```

6.2 Gradient Check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-1.

[19]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(260)
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)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-1
```

```
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('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
W1 max relative error: 3.677126e-01
W2 max relative error: 1.772601e-02
W3 max relative error: 4.013935e-05
b1 max relative error: 9.164054e-05
b2 max relative error: 1.727703e-06
b3 max relative error: 1.143528e-09
```

6.3 Overfit Small Data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

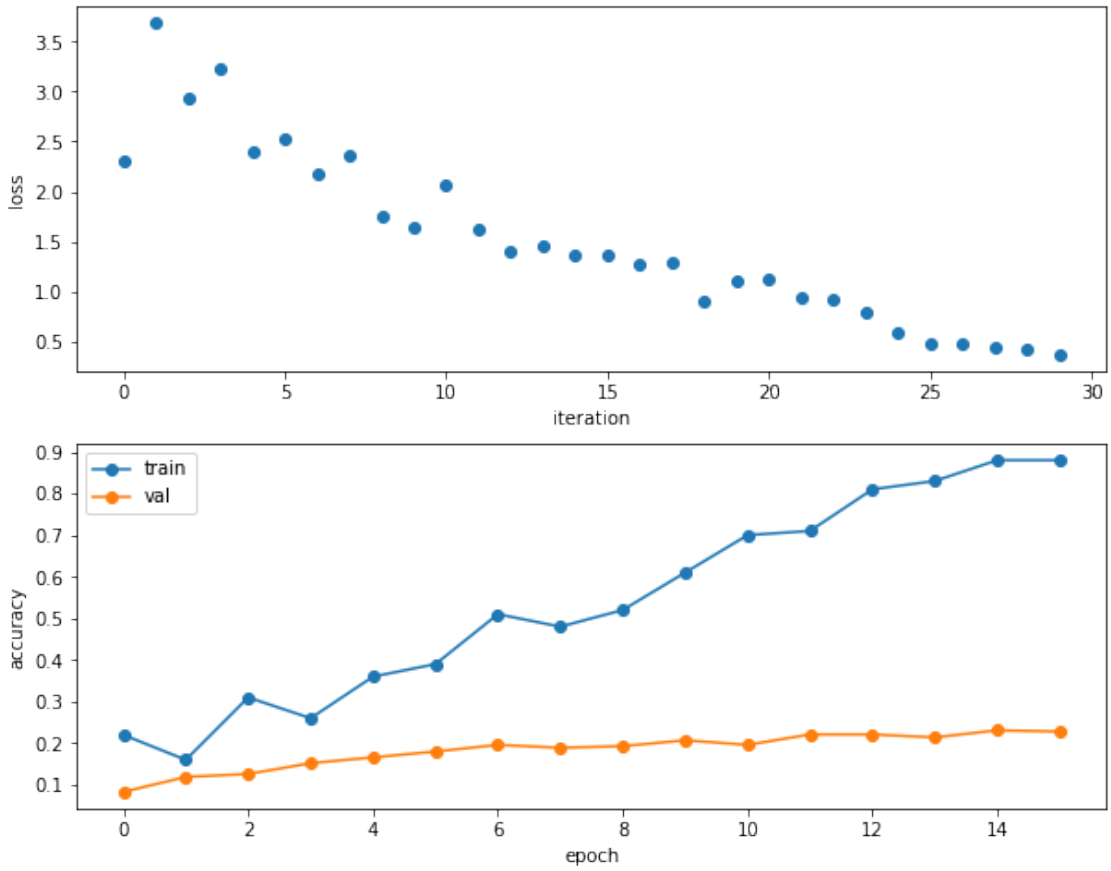
[20]:
```
np.random.seed(260)

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=15,
    batch_size=50,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=1
)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.299837
(Epoch 0 / 15) train acc: 0.220000; val_acc: 0.083000
```

```
(Iteration 2 / 30) loss: 3.695521
(Epoch 1 / 15) train acc: 0.160000; val_acc: 0.119000
(Iteration 3 / 30) loss: 2.939817
(Iteration 4 / 30) loss: 3.237984
(Epoch 2 / 15) train acc: 0.310000; val_acc: 0.126000
(Iteration 5 / 30) loss: 2.392279
(Iteration 6 / 30) loss: 2.529508
(Epoch 3 / 15) train acc: 0.260000; val_acc: 0.152000
(Iteration 7 / 30) loss: 2.167502
(Iteration 8 / 30) loss: 2.361092
(Epoch 4 / 15) train acc: 0.360000; val_acc: 0.166000
(Iteration 9 / 30) loss: 1.750063
(Iteration 10 / 30) loss: 1.638533
(Epoch 5 / 15) train acc: 0.390000; val_acc: 0.180000
(Iteration 11 / 30) loss: 2.058860
(Iteration 12 / 30) loss: 1.621344
(Epoch 6 / 15) train acc: 0.510000; val_acc: 0.196000
(Iteration 13 / 30) loss: 1.404665
(Iteration 14 / 30) loss: 1.448090
(Epoch 7 / 15) train acc: 0.480000; val_acc: 0.189000
(Iteration 15 / 30) loss: 1.366978
(Iteration 16 / 30) loss: 1.367789
(Epoch 8 / 15) train acc: 0.520000; val_acc: 0.193000
(Iteration 17 / 30) loss: 1.268402
(Iteration 18 / 30) loss: 1.279374
(Epoch 9 / 15) train acc: 0.610000; val_acc: 0.207000
(Iteration 19 / 30) loss: 0.900562
(Iteration 20 / 30) loss: 1.104695
(Epoch 10 / 15) train acc: 0.700000; val_acc: 0.196000
(Iteration 21 / 30) loss: 1.121203
(Iteration 22 / 30) loss: 0.927204
(Epoch 11 / 15) train acc: 0.710000; val_acc: 0.221000
(Iteration 23 / 30) loss: 0.924401
(Iteration 24 / 30) loss: 0.795979
(Epoch 12 / 15) train acc: 0.810000; val_acc: 0.221000
(Iteration 25 / 30) loss: 0.576027
(Iteration 26 / 30) loss: 0.466703
(Epoch 13 / 15) train acc: 0.830000; val_acc: 0.214000
(Iteration 27 / 30) loss: 0.465192
(Iteration 28 / 30) loss: 0.432180
(Epoch 14 / 15) train acc: 0.880000; val_acc: 0.231000
(Iteration 29 / 30) loss: 0.408458
(Iteration 30 / 30) loss: 0.369085
(Epoch 15 / 15) train acc: 0.880000; val_acc: 0.228000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

[21]:
```
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()
```



6.4 Train the Network

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

[22]:
```
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()
```

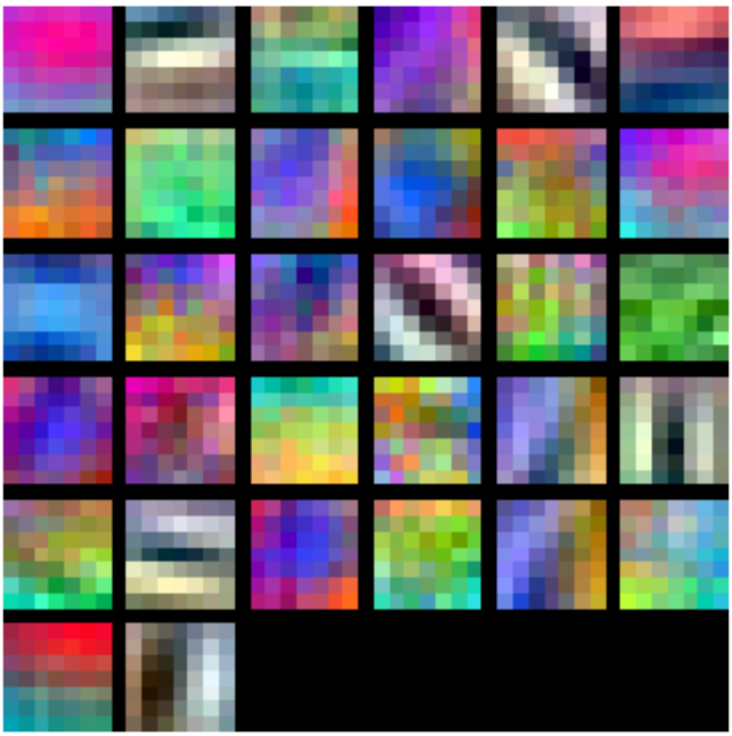
```
(Iteration 1 / 980) loss: 2.304627
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.400579
(Iteration 41 / 980) loss: 2.273675
(Iteration 61 / 980) loss: 1.782403
(Iteration 81 / 980) loss: 1.894766
(Iteration 101 / 980) loss: 1.679100
(Iteration 121 / 980) loss: 1.751259
(Iteration 141 / 980) loss: 1.579113
(Iteration 161 / 980) loss: 1.880709
(Iteration 181 / 980) loss: 1.795410
(Iteration 201 / 980) loss: 2.377685
(Iteration 221 / 980) loss: 1.653197
(Iteration 241 / 980) loss: 1.750863
(Iteration 261 / 980) loss: 1.762987
(Iteration 281 / 980) loss: 1.952407
(Iteration 301 / 980) loss: 1.422582
(Iteration 321 / 980) loss: 1.615992
(Iteration 341 / 980) loss: 1.711509
(Iteration 361 / 980) loss: 1.341072
(Iteration 381 / 980) loss: 1.720951
(Iteration 401 / 980) loss: 1.779001
(Iteration 421 / 980) loss: 1.625151
(Iteration 441 / 980) loss: 1.611447
(Iteration 461 / 980) loss: 1.711153
(Iteration 481 / 980) loss: 1.608832
(Iteration 501 / 980) loss: 1.732382
(Iteration 521 / 980) loss: 1.624103
(Iteration 541 / 980) loss: 1.901537
(Iteration 561 / 980) loss: 1.508895
(Iteration 581 / 980) loss: 1.655254
(Iteration 601 / 980) loss: 1.756955
(Iteration 621 / 980) loss: 1.676070
```

```
(Iteration 641 / 980) loss: 1.280857
(Iteration 661 / 980) loss: 1.609022
(Iteration 681 / 980) loss: 1.526835
(Iteration 701 / 980) loss: 1.249036
(Iteration 721 / 980) loss: 1.488000
(Iteration 741 / 980) loss: 1.646981
(Iteration 761 / 980) loss: 1.320006
(Iteration 781 / 980) loss: 1.645626
(Iteration 801 / 980) loss: 1.530392
(Iteration 821 / 980) loss: 1.685753
(Iteration 841 / 980) loss: 1.482879
(Iteration 861 / 980) loss: 1.595603
(Iteration 881 / 980) loss: 1.743055
(Iteration 901 / 980) loss: 1.348698
(Iteration 921 / 980) loss: 1.335930
(Iteration 941 / 980) loss: 1.874940
(Iteration 961 / 980) loss: 1.586424
(Epoch 1 / 1) train acc: 0.449000; val_acc: 0.420000
```

6.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

[ ]:
```
from cs260.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



7 Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully connected networks. As proposed in the original paper (link in `BatchNormalization.ipynb`), batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

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 needs to accept inputs of shape `(N, C, H, W)` and produce 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.

If the feature map was produced using convolutions, then we expect every feature channel's statistics e.g. mean, variance to be relatively consistent both between different images, and different locations within the same image -- after all, every feature channel is produced by the same convolutional filter! Therefore, spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over the minibatch dimension `N` as well the spatial dimensions `H` and `W`.

[1] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

8 Spatial Batch Normalization: Forward Pass

In the file `cs260/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[23]:
```
np.random.seed(260)

# 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:  [10.1270009  10.34995476 10.22876718]
  stds:  [4.34690483 3.86000959 3.80904389]
After spatial batch normalization:
  shape:  (2, 3, 4, 5)
  means:  [-4.27435864e-16  1.52655666e-16 -8.88178420e-17]
  stds:  [0.99999974 0.99999966 0.99999966]
After spatial batch normalization (nontrivial gamma, beta):
  shape:  (2, 3, 4, 5)
  means:  [6. 7. 8.]
  stds:  [2.99999921 3.99999866 4.99999828]
```

[24]:
```
np.random.seed(260)

# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [0.04302458 0.08924234 0.05991434 0.03833107]
  stds:  [1.01576135 0.99228064 0.98902125 0.9988344 ]
```