

Modular neural nets

In the previous homework, we implemented modular neural networks for a two-layer neural network with fully connected layers. Now, we will do the same for convolutional layers. Once again, the benefit of modular designs is that we can snap together different types of layers and loss functions in order to quickly experiment with different architectures.

```
In [45]: # As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from cs231n.conv_layers import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

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

Convolution layer: forward naive

Implement the function `conv_forward_naive` in the file `cs231n/conv_layers.py`.

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:

```
In [46]: 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
```

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.

```
In [47]: from PIL import Image
kitten, puppy = np.array(Image.open('kitten.jpg')), np.array(Image.open('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
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = np.array(Image.fromarray(puppy).resize((img_size, img_size))).transpose((2, 0, 1))
x[1, :, :, :] = np.array(Image.fromarray(kitten_cropped).resize((img_size, img_size))).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})

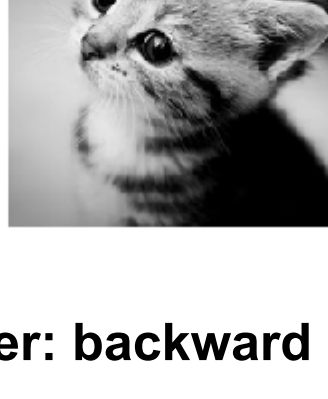
def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```


Original image



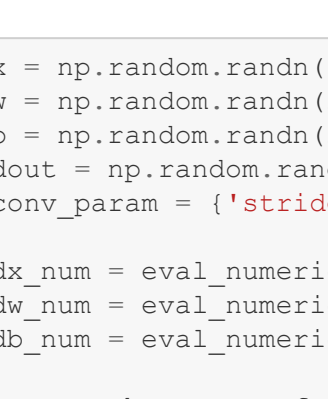
Grayscale



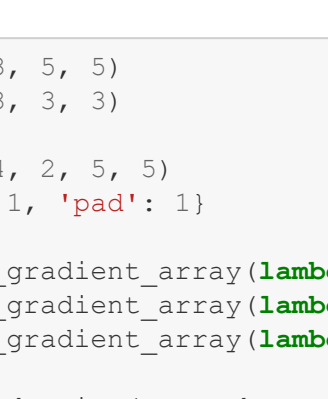
Edges



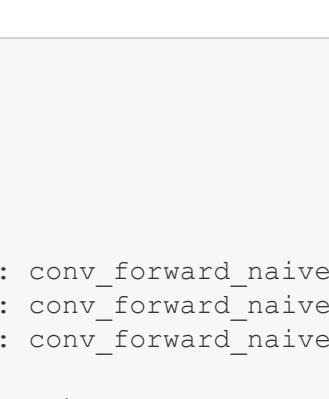
Original image



Grayscale



Edges



Convolution layer: backward naive

Next you need to implement the function `conv_backward_naive` in the file `cs231n/conv_layers.py`. As usual, we will check your implementation with numeric gradient checking.

```
In [48]: 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}

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: 9.682029025656176e-10
dw error: 5.431767655780389e-10
db error: 2.929860525174844e-11
```

Max pooling layer: forward naive

The last layer we need for a basic convolutional neural network is the max pooling layer. First implement the forward pass in the function `max_pool_forward_naive` in the file `cs231n/conv_layers.py`.

```
In [49]: x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[[-0.26315789, -0.24842105],
                           [-0.20421053, -0.18947368]],
                           [[-0.14526316, -0.13052632],
                           [-0.08631579, -0.07157895]],
                           [[-0.02736842, -0.01263158],
                           [ 0.03157895,  0.04631579]]],
                           [[[-0.09052632,  0.10526316],
                           [ 0.14947368,  0.16421053]],
                           [[ 0.20842105,  0.22315789],
                           [ 0.26736842,  0.28210526]],
                           [[ 0.32631579,  0.34105263],
                           [ 0.38526316,  0.4          ]]]]])

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

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

Max pooling layer: backward naive

Implement the backward pass for a max pooling layer in the function `max_pool_backward_naive` in the file `cs231n/conv_layers.py`. As always we check the correctness of the backward pass using numerical gradient checking.

```
In [50]: 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: 3.2756192835058297e-12
```

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 `cs231n/fast_layers.py`.

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

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

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 receives 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:

```
In [51]: from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

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

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

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

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

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

Testing conv_forward_fast:
Naive: 0.09873s
Fast: 0.01597s
Speedup: 6.187587x
Difference: 3.104202640264915e-11

Testing conv_backward_fast:
Naive: 0.221409s
Fast: 0.025931s
Speedup: 8.538363x
dx difference: 3.008014534569333e-12
dw difference: 1.0814979782973717e-13
db difference: 0.0
```

```
In [52]: from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

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

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

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

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

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('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: 0.014960s
fast: 0.004987s
speedup: 3.000143x
difference: 0.0

Testing pool_backward_fast:
Naive: 3.017929s
fast: 0.012967s
speedup: 232.732942x
dx difference: 0.0
```

Sandwich layers

There are a couple common layer "sandwiches" that frequently appear in ConvNets. For example convolutional layers are frequently followed by ReLU and pooling, and affine layers are frequently followed by ReLU. To make it more convenient to use these common patterns, we have defined several convenience layers in the file `cs231n/layer_utils.py`. Lets grad-check them to make sure that they work correctly:

```
In [53]: from cs231n.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)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_param, pool_param)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_param, pool_param)
)

print('Testing conv_relu_pool_forward:')
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_forward:
dx error: 1.5359714066853973e-08
dw error: 6.730359554066423e-10
db error: 3.0926810572024394e-11
```

```
In [54]: from cs231n.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_forward:')
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_forward:
dx error: 1.6412906626008716e-09
dw error: 1.8805714936176815e-09
db error: 2.183516243068678e-11
```

```
In [55]: from cs231n.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, dout)

print('Testing affine_relu_forward:')
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 affine_relu_forward:
dx error: 1.6226945537770296e-09
dw error: 3.5321483340718618e-09
db error: 1.8928900002169046e-11
```

```
In [ ]:
```


Train a ConvNet!

We now have a generic solver and a bunch of modularized layers. It's time to put it all together, and train a ConvNet to recognize the classes in CIFAR-10. In this notebook we will walk you through training a simple two-layer ConvNet and then set you free to build the best net that you can to perform well on CIFAR-10.

Open up the file `cs231n/classifiers/convnet.py`; you will see that the `two_layer_convnet` function computes the loss and gradients for a two-layer ConvNet. Note that this function uses the "sandwich" layers defined in `cs231n/layer_utils.py`.

```
In [3]: # As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifier_trainer import ClassifierTrainer
from cs231n.gradient_check import eval_numerical_gradient
from cs231n.classifiers.convnet import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
In [4]: from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Transpose so that channels come first
    X_train = X_train.transpose(0, 3, 1, 2).copy()
    X_val = X_val.transpose(0, 3, 1, 2).copy()
    X_test = X_test.transpose(0, 3, 1, 2).copy()

    return X_train, y_train, X_val, y_val, X_test, y_test

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

Train data shape: (49000, 3, 32, 32)
Train labels shape: (49000,)
Validation data shape: (1000, 3, 32, 32)
Validation labels shape: (1000,)
Test data shape: (1000, 32, 32, 3)
Test labels shape: (1000,)

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 this should go up.

```
In [5]: model = init_two_layer_convnet()

X = np.random.randn(100, 3, 32, 32)
y = np.random.randint(10, size=100)

loss, _ = two_layer_convnet(X, model, y, reg=0)

# Sanity check: Loss should be about log(10) = 2.3026
print('Sanity check loss (no regularization): ', loss)

# Sanity check: Loss should go up when you add regularization
loss, _ = two_layer_convnet(X, model, y, reg=1)
print('Sanity check loss (with regularization): ', loss)
```

Sanity check loss (no regularization): 2.3024942537348783
Sanity check loss (with regularization): 2.3443364674260807

Gradient check

After the loss looks reasonable, you should always 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 artificial data and a small number of neurons at each layer.

```
In [6]: num_inputs = 2
input_shape = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_shape)
y = np.random.randint(num_classes, size=num_inputs)

model = init_two_layer_convnet(num_filters=3, filter_size=3, input_shape=input_shape)
loss, grads = two_layer_convnet(X, model, y)
for param_name in sorted(grads):
    f = lambda _: two_layer_convnet(X, model, y)[0]
    param_grad_num = eval_numerical_gradient(f, model[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: 1.416301e-06
W2 max relative error: 4.636411e-06
b1 max relative error: 1.130980e-08
b2 max relative error: 8.226398e-10
```

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.

```
In [7]: # Use a two-layer ConvNet to overfit 50 training examples.

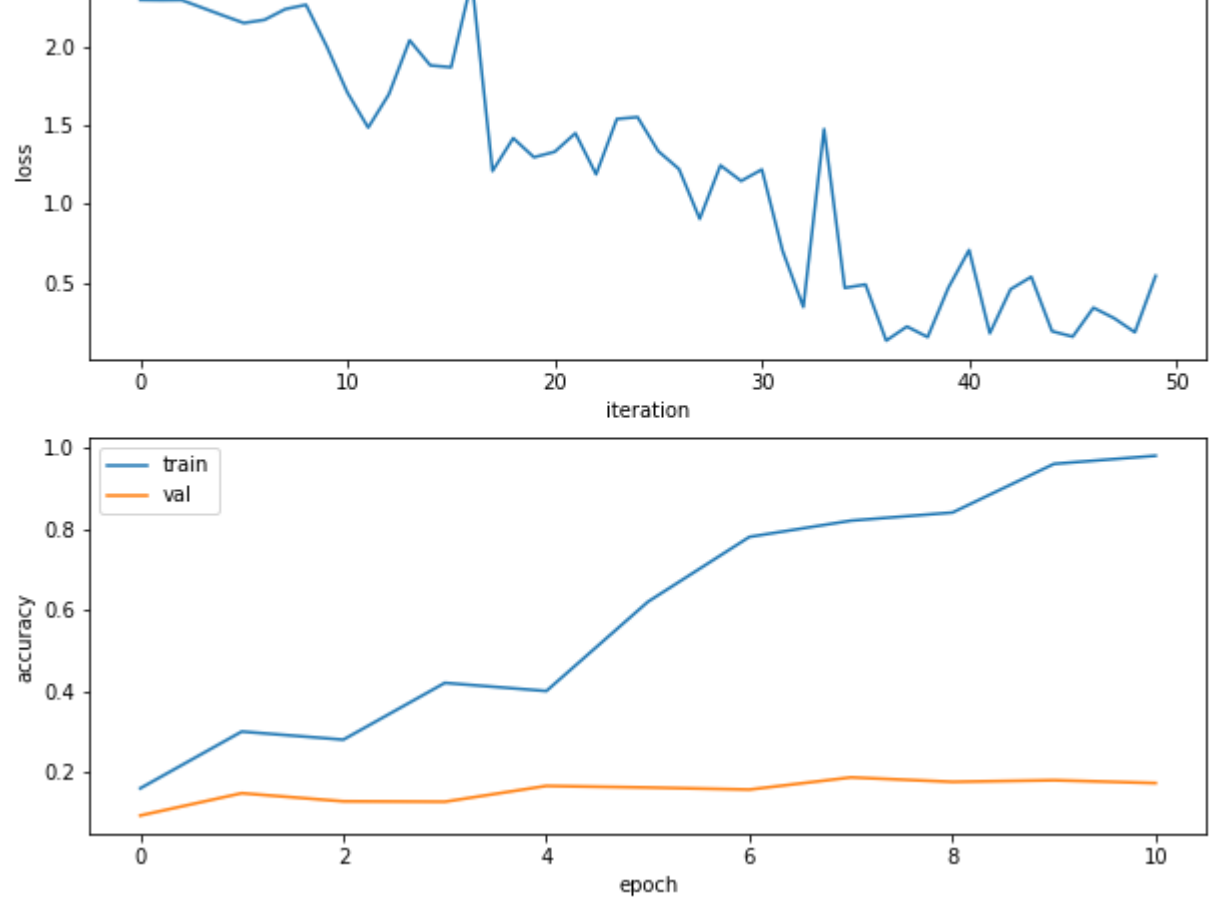
model = init_two_layer_convnet()
trainer = ClassifierTrainer()
best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
    X_train[:50], y_train[:50], X_val, y_val, model, two_layer_convnet,
    reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=10, num_epochs=10,
    verbose=True)

starting iteration 0
Finished epoch 0 / 10: cost 2.293451, train: 0.160000, val 0.093000, lr 1.000000e-04
Finished epoch 1 / 10: cost 2.194749, train: 0.300000, val 0.148000, lr 9.500000e-05
Finished epoch 2 / 10: cost 1.998509, train: 0.280000, val 0.128000, lr 9.025000e-05
starting iteration 10
Finished epoch 3 / 10: cost 1.878304, train: 0.420000, val 0.127000, lr 8.573750e-05
Finished epoch 4 / 10: cost 1.295153, train: 0.400000, val 0.166000, lr 8.145062e-05
starting iteration 20
Finished epoch 5 / 10: cost 1.550190, train: 0.620000, val 0.162000, lr 7.737809e-05
Finished epoch 6 / 10: cost 1.144751, train: 0.780000, val 0.157000, lr 7.350919e-05
starting iteration 30
Finished epoch 7 / 10: cost 0.466211, train: 0.820000, val 0.187000, lr 6.983373e-05
Finished epoch 8 / 10: cost 0.467357, train: 0.840000, val 0.176000, lr 6.634204e-05
starting iteration 40
Finished epoch 9 / 10: cost 0.191292, train: 0.960000, val 0.180000, lr 6.302494e-05
Finished epoch 10 / 10: cost 0.542685, train: 0.980000, val 0.173000, lr 5.987369e-05
finished optimization. best validation accuracy: 0.187000
```

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

```
In [8]: plt.subplot(2, 1, 1)
plt.plot(loss_history)
plt.xlabel('iteration')
plt.ylabel('loss')

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



Train the net

Once the above works, training the net is the next thing to try. You can set the `acc_frequency` parameter to change the frequency at which the training and validation set accuracies are tested. If your parameters are set properly, you should see the training and validation accuracy start to improve within a hundred iterations, and you should be able to train a reasonable model with just one epoch.

Using the parameters below you should be able to get around 50% accuracy on the validation set.

```
In [11]: model = init_two_layer_convnet(filter_size=7)
trainer = ClassifierTrainer()
best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
    X_train, y_train, X_val, y_val, model, two_layer_convnet,
    reg=0.1, momentum=0.9, learning_rate=0.0001, batch_size=256, num_epochs=1,
    acc_frequency=50, verbose=True)

starting iteration 0
Finished epoch 0 / 1: cost 2.317788, train: 0.084000, val 0.098000, lr 1.000000e-04
starting iteration 10
starting iteration 20
starting iteration 30
starting iteration 40
starting iteration 50
Finished epoch 0 / 1: cost 1.789126, train: 0.375000, val 0.389000, lr 1.000000e-04
starting iteration 60
starting iteration 70
starting iteration 80
starting iteration 90
starting iteration 100
Finished epoch 0 / 1: cost 1.636210, train: 0.463000, val 0.474000, lr 1.000000e-04
starting iteration 110
starting iteration 120
starting iteration 130
starting iteration 140
starting iteration 150
Finished epoch 0 / 1: cost 1.547649, train: 0.478000, val 0.489000, lr 1.000000e-04
starting iteration 160
starting iteration 170
starting iteration 180
starting iteration 190
Finished epoch 1 / 1: cost 1.379095, train: 0.514000, val 0.520000, lr 9.500000e-05
finished optimization. best validation accuracy: 0.520000
```

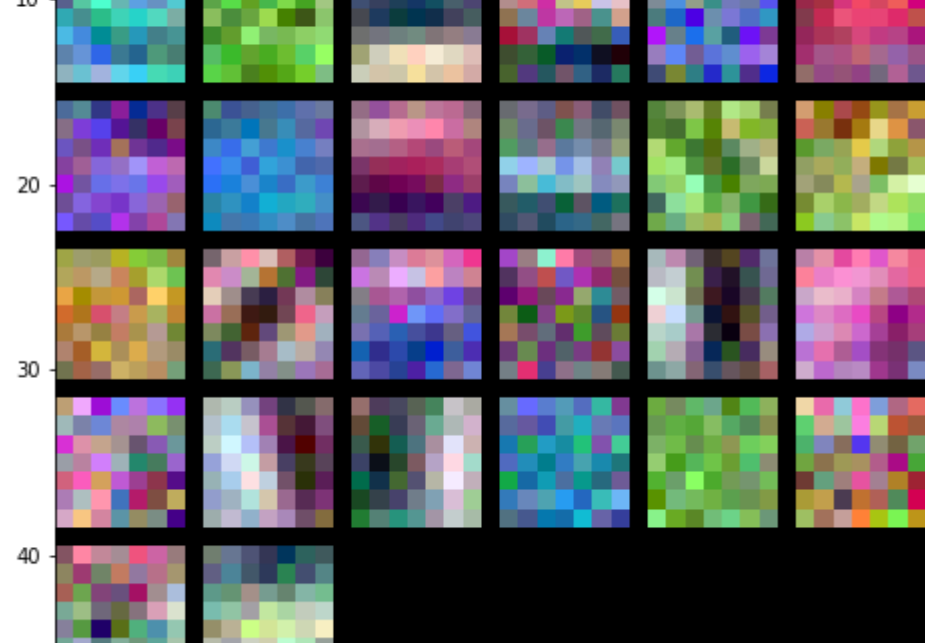
Visualize weights

We can visualize the convolutional weights from the first layer. If everything worked properly, these will usually be edges and blobs of various colors and orientations.

```
In [12]: from cs231n.vis_utils import visualize_grid

grid = visualize_grid(best_model['W1']).transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
```

Out[12]: <matplotlib.image.AxesImage at 0x7f1e3a71ef60>



PyTorch data

PyTorch comes with a nice paradigm for dealing with data which we'll use here. A PyTorch `Dataset` knows where to find data in its raw form (files on disk) and how to load individual examples into Python datastructures. A PyTorch `DataLoader` takes a dataset and offers a variety of ways to sample batches from that dataset.

Take a moment to browse through the `CIFAR10 Dataset` in `2_pytorch/cifar10.py`, read the `DataLoader` documentation linked above, and see how these are used in the section of `train.py` that loads data. Note that in the first part of the homework we subtracted a mean CIFAR10 image from every image before feeding it in to our models. Here we subtract a constant color instead. Both methods are seen in practice and work equally well.

PyTorch provides lots of vision datasets which can be imported directly from `torchvision.datasets`. Also see `torchtext` for natural language datasets.

ConvNet Classifier in PyTorch

In PyTorch Deep Learning building blocks are implemented in the neural network module `torch.nn` (usually imported as `nn`). A PyTorch model is typically a subclass of `nn.Module` and thereby gains a multitude of features. Because your logistic regressor is an `nn.Module` all of its parameters and sub-modules are accessible through the `.parameters()` and `.modules()` methods.

Now implement a ConvNet classifier by filling in the marked sections of `models/convnet.py`.

The main driver for this question is `train.py`. It reads arguments and model hyperparameter from the command line, loads CIFAR10 data and the specified model (in this case, softmax). Using the optimizer initialized with appropriate hyperparameters, it trains the model and reports performance on test data.

Complete the following couple of sections in `train.py`:

1. Initialize an optimizer from the `torch.optim` package
2. Update the parameters in model using the optimizer initialized above

At this point all of the components required to train the softmax classifier are complete for the softmax classifier. Now run

```
$ run_convnet.sh
```

to train a model and save it to `convnet.pt`. This will also produce a `convnet.log` file which contains training details which we will visualize below.

Note: You may want to adjust the hyperparameters specified in `run_convnet.sh` to get reasonable performance.

Visualizing the PyTorch model

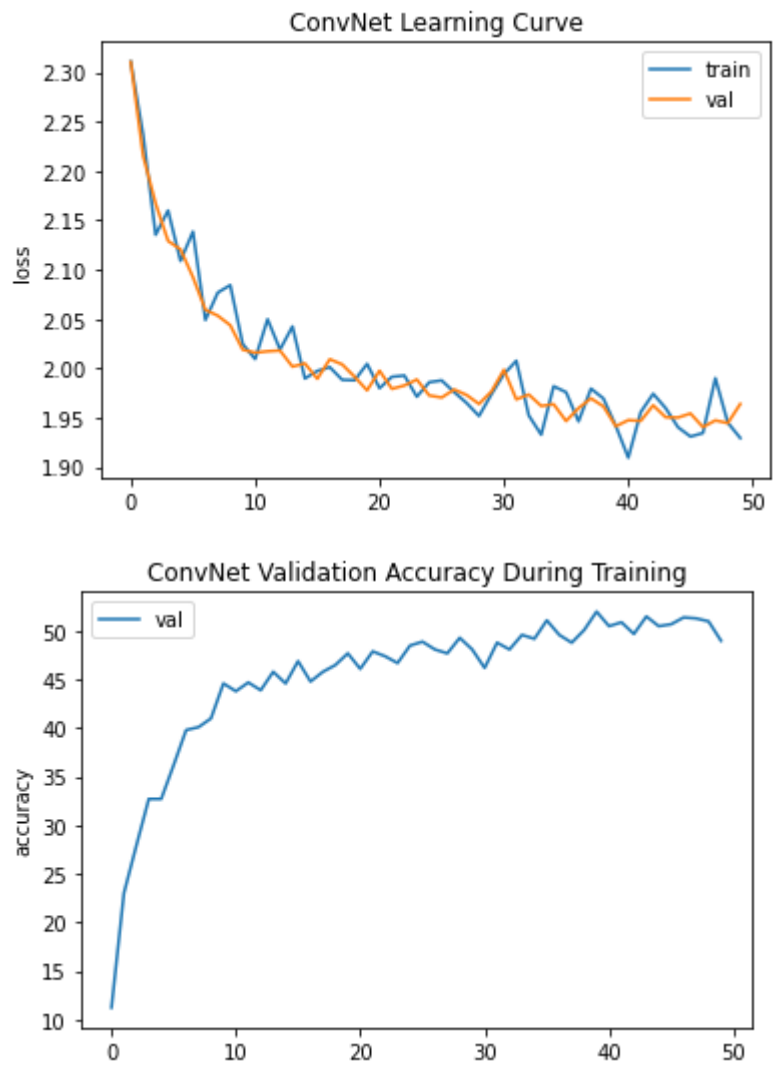
```
In [1]: # Assuming that you have completed training the classifier, let us plot the training loss vs. iteration.
        # This is an
        # example to show a simple way to log and plot data from PyTorch.

        # we need matplotlib to plot the graphs for us!
import matplotlib
# This is needed to save images
matplotlib.use('Agg')
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [4]: # Parse the train and val losses one line at a time.
import re
# regexes to find train and val losses on a line
float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
train_loss_re = re.compile('.*Train Loss: ({})' .format(float_regex))
val_loss_re = re.compile('.*Val Loss: ({})' .format(float_regex))
val_acc_re = re.compile('.*Val Acc: ({})' .format(float_regex))
# extract one loss for each logged iteration
train_losses = []
val_losses = []
val_accs = []
# NOTE: You may need to change this file name.
with open('convnet.log', 'r') as f:
    for line in f:
        train_match = train_loss_re.match(line)
        val_match = val_loss_re.match(line)
        val_acc_match = val_acc_re.match(line)
        if train_match:
            train_losses.append(float(train_match.group(1)))
        if val_match:
            val_losses.append(float(val_match.group(1)))
        if val_acc_match:
            val_accs.append(float(val_acc_match.group(1)))
```

```
In [5]: fig = plt.figure()
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.title('ConvNet Learning Curve')
plt.ylabel('loss')
plt.legend()
fig.savefig('convnet_lossvstrain.png')

fig = plt.figure()
plt.plot(val_accs, label='val')
plt.title('ConvNet Validation Accuracy During Training')
plt.ylabel('accuracy')
plt.legend()
fig.savefig('convnet_valaccuracy.png')
```



```
In [ ]:
```

Visualizing the trained filters

```
In [59]: # some startup!
import numpy as np
import matplotlib
# This is needed to save images
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import torch
```

```
In [60]: # load the model saved by train.py
# This will be an instance of models.convnet.CNN.
# NOTE: You may need to change this file name.
convnet_model = torch.load('convnet.pt')
```

```
In [61]: # collect all the weights
w = None
#####
# TODO: Extract the weight matrix (without bias) from convnet_model, convert
# it to a numpy array with shape (10, 32, 32, 3), and assign this array to w.
# The first dimension should be for channels, then height, width, and color.
# This step depends on how you implemented models.convnet.CNN.
#####
w = convnet_model.conv1.weight.detach().reshape(10,3,7,7).permute(0,2,3,1).numpy()
#####
#                                     END OF YOUR CODE                                     #
#####
# obtain min,max to normalize
w_min, w_max = np.min(w), np.max(w)
# classes
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
# init figure
fig = plt.figure(figsize=(6,6))
for i in range(10):
    wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
    # subplot is (2,5) as ten filters are to be visualized
    fig.add_subplot(2,5,i+1).imshow(wimg.astype('uint8'))
# save fig!
fig.savefig('convnet_filt.png')
print('figure saved')
```

figure saved

```
In [62]: # vis_utils.py has helper code to view multiple filters in single image. Use this to visuzlize
# neural network.
# import vis_utils
from vis_utils import visualize_grid
# saving the weights is now as simple as:
plt.imsave('convnet_gridfilt.png',visualize_grid(w, padding=3).astype('uint8'))
# padding is the space between images. Make sure that w is of shape: (N,H,W,C)
print('figure saved as a grid!')
```

figure saved as a grid!

```
In [ ]:
```