Convolutional neural networks

In this notebook, we'll put together our convolutional layers to implement a 3-layer CNN. Then, we'll ask you to implement a CNN that can achieve > 65% validation error on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

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

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

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

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

In [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))))
```

In [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,)
```

Three layer CNN

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

The architecture of this CNN will be:

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

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

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

In [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: 6.429047084857119e-05
W2 max relative error: 0.02113108389377925
W3 max relative error: 0.00015875456098393367
b1 max relative error: 9.579599500253946e-06
b2 max relative error: 2.801110792370736e-07
b3 max relative error: 1.1270610452497654e-09
```

Overfit small dataset

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

In [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()
```

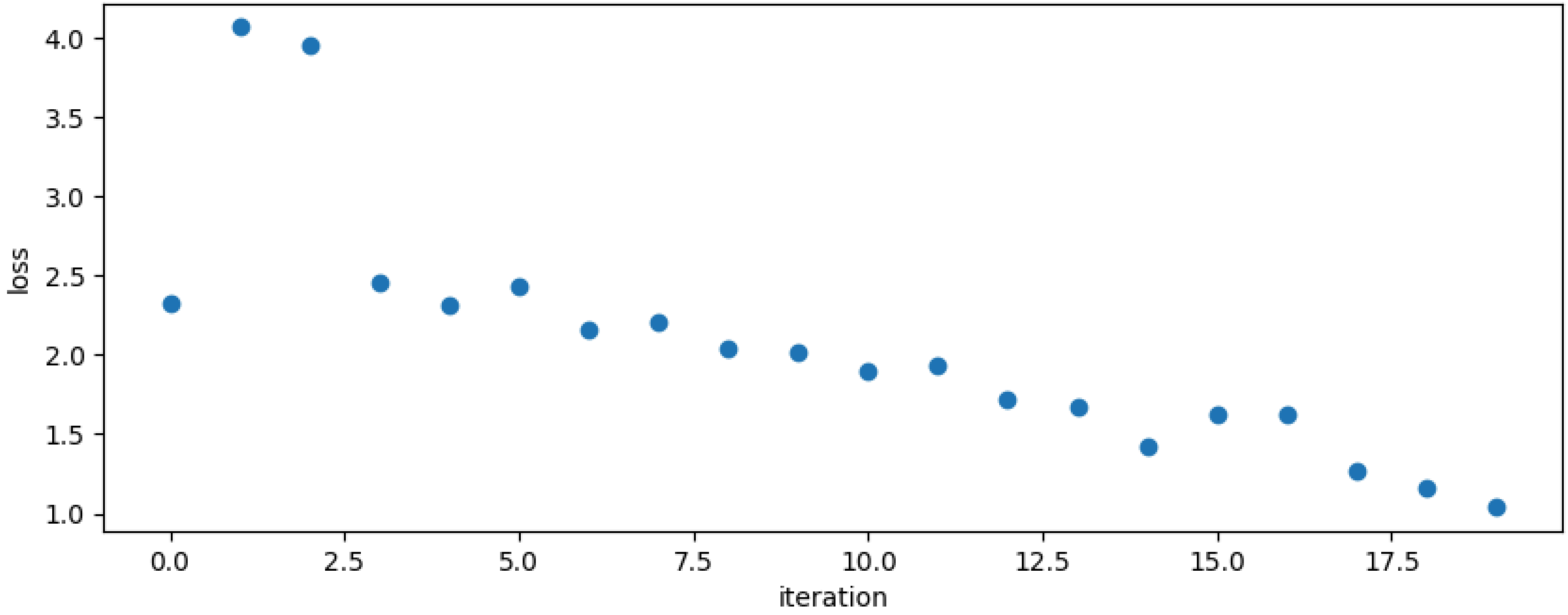
```
(Iteration 1 / 20) loss: 2.319779
(Epoch 0 / 10) train acc: 0.160000; val_acc: 0.119000
(Iteration 2 / 20) loss: 4.063094
(Epoch 1 / 10) train acc: 0.130000; val_acc: 0.106000
(Iteration 3 / 20) loss: 3.955578
(Iteration 4 / 20) loss: 2.455073
(Epoch 2 / 10) train acc: 0.210000; val_acc: 0.116000
(Iteration 5 / 20) loss: 2.311101
(Iteration 6 / 20) loss: 2.434196
(Epoch 3 / 10) train acc: 0.310000; val_acc: 0.115000
(Iteration 7 / 20) loss: 2.155987
(Iteration 8 / 20) loss: 2.199559
(Epoch 4 / 10) train acc: 0.330000; val_acc: 0.130000
(Iteration 9 / 20) loss: 2.043093
(Iteration 10 / 20) loss: 2.012471
(Epoch 5 / 10) train acc: 0.420000; val_acc: 0.143000
(Iteration 11 / 20) loss: 1.899885
(Iteration 12 / 20) loss: 1.932975
(Epoch 6 / 10) train acc: 0.480000; val_acc: 0.145000
(Iteration 13 / 20) loss: 1.718187
(Iteration 14 / 20) loss: 1.674054
(Epoch 7 / 10) train acc: 0.500000; val_acc: 0.174000
(Iteration 15 / 20) loss: 1.426449
(Iteration 16 / 20) loss: 1.623057
(Epoch 8 / 10) train acc: 0.560000; val_acc: 0.164000
(Iteration 17 / 20) loss: 1.624247
(Iteration 18 / 20) loss: 1.269783
(Epoch 9 / 10) train acc: 0.720000; val_acc: 0.201000
(Iteration 19 / 20) loss: 1.157586
(Iteration 20 / 20) loss: 1.042304
(Epoch 10 / 10) train acc: 0.750000; val_acc: 0.194000
```

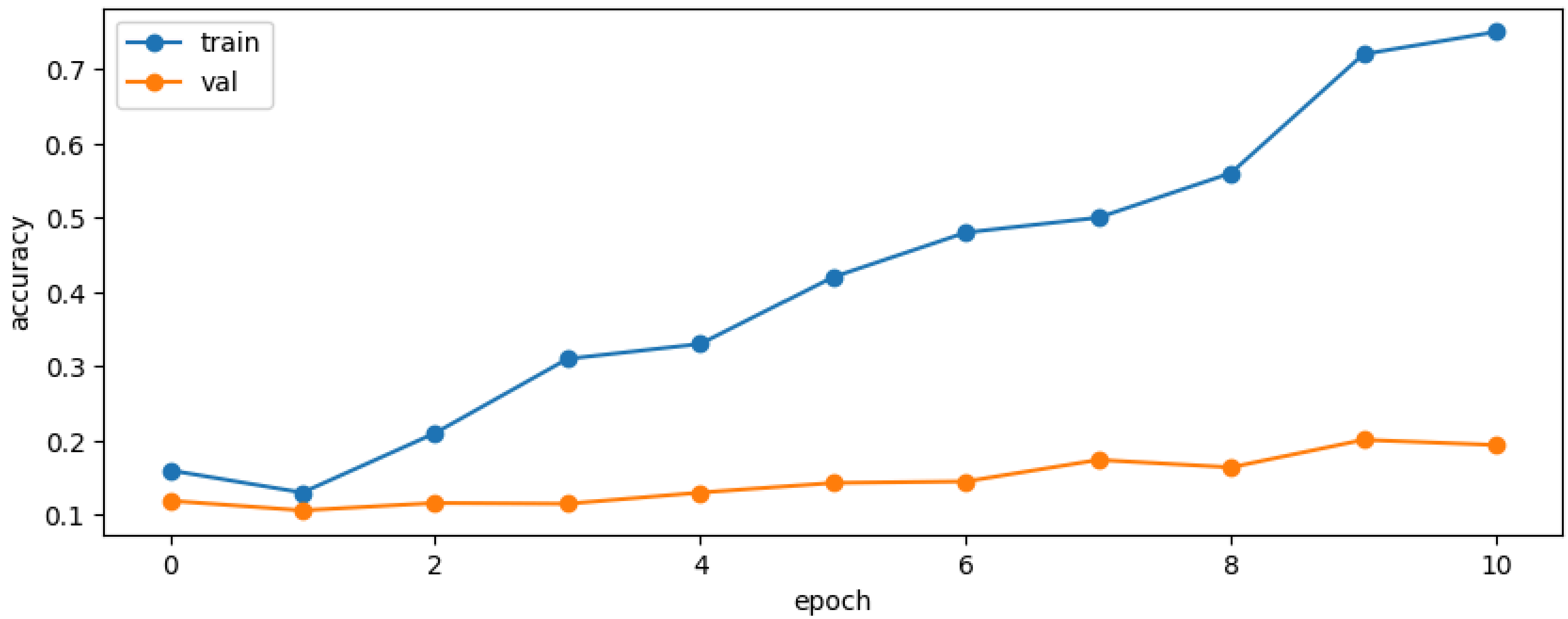
In [5]:

```python
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()
```


4.0
3.5
3.0
2.5
2.0
1.5
1.0
loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
iteration


train
val
0.7
0.6
0.5
0.4
0.3
0.2
0.1
accuracy
0
2
4
6
8
10
epoch

Train the network

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

In [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: 2.305011
(Epoch 0 / 1) train acc: 0.093000; val_acc: 0.098000
(Iteration 21 / 980) loss: 2.219201
(Iteration 41 / 980) loss: 2.259111
(Iteration 61 / 980) loss: 2.419081
(Iteration 81 / 980) loss: 1.932590
(Iteration 101 / 980) loss: 1.939837
(Iteration 121 / 980) loss: 1.989934
(Iteration 141 / 980) loss: 2.124409
(Iteration 161 / 980) loss: 1.779174
(Iteration 181 / 980) loss: 1.970763
(Iteration 201 / 980) loss: 1.722760
(Iteration 221 / 980) loss: 1.636082
(Iteration 241 / 980) loss: 1.949602
(Iteration 261 / 980) loss: 1.934016
(Iteration 281 / 980) loss: 1.616354
(Iteration 301 / 980) loss: 1.633906
(Iteration 321 / 980) loss: 1.822530
(Iteration 341 / 980) loss: 1.838734
(Iteration 361 / 980) loss: 1.721726
(Iteration 381 / 980) loss: 1.880330
(Iteration 401 / 980) loss: 1.348953
(Iteration 421 / 980) loss: 1.598387
(Iteration 441 / 980) loss: 1.599144
(Iteration 461 / 980) loss: 1.889709
(Iteration 481 / 980) loss: 1.546496
(Iteration 501 / 980) loss: 1.657716
(Iteration 521 / 980) loss: 1.378137
(Iteration 541 / 980) loss: 1.314164
(Iteration 561 / 980) loss: 1.480258
(Iteration 581 / 980) loss: 1.916595
(Iteration 601 / 980) loss: 1.886212
(Iteration 621 / 980) loss: 1.694972
(Iteration 641 / 980) loss: 1.661934
(Iteration 661 / 980) loss: 1.416175
(Iteration 681 / 980) loss: 1.535042
(Iteration 701 / 980) loss: 1.843636
(Iteration 721 / 980) loss: 1.430453
(Iteration 741 / 980) loss: 1.428181
(Iteration 761 / 980) loss: 1.773698
(Iteration 781 / 980) loss: 1.633876
```

```
(Iteration 801 / 980) loss: 1.541591
(Iteration 821 / 980) loss: 1.592416
(Iteration 841 / 980) loss: 1.889209
(Iteration 861 / 980) loss: 1.572581
(Iteration 881 / 980) loss: 1.771842
(Iteration 901 / 980) loss: 1.665086
(Iteration 921 / 980) loss: 1.524211
(Iteration 941 / 980) loss: 1.574793
(Iteration 961 / 980) loss: 1.435574
(Epoch 1 / 1) train acc: 0.457000; val_acc: 0.486000
```

Get > 65% validation accuracy on CIFAR-10.

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

Things you should try:

- Filter size: Above we used 7x7; but VGGNet and onwards showed stacks of 3x3 filters are good.
- Number of filters: Above we used 32 filters. Do more or fewer do better?
- Batch normalization: Try adding spatial batch normalization after convolution layers and vanilla batch normalization aafter affine layers. Do your networks train faster?
- Network architecture: Can a deeper CNN do better? Consider these architectures:
  - [conv-relu-pool]xN - conv - relu - [affine]xM - [softmax or SVM]
  - [conv-relu-pool]XN - [affine]XM - [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN - [affine]xM - [softmax or SVM]

Tips for training

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

- If the parameters are working well, you should see improvement within a few hundred iterations

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

In [ ]:

In [ ]:

In [7]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #
model = ThreeLayerConvNet(
    filter_size=3,  # decrease filter size
    num_filters=70,  # increase number of filters
    weight_scale=0.01,  # increase weight scale
    use_batchnorm=True,  # enable batchnorm
    hidden_dim=500,
    reg=0.001,
)

solver = Solver(
    model,
    data,
    num_epochs=5,  # increase number of epochs
    batch_size=512,  # increase batch size
    lr_decay=0.9,  # add learning rate decay
    update_rule="adam",
    optim_config={
        "learning_rate": 1e-3,
    },
    verbose=True,
    print_every=20,
)

solver.train()
```

```
y_val_max = np.argmax(model.loss(data["X_val"]), axis=1)
y_test_max = np.argmax(model.loss(data["X_test"]), axis=1)
print("Validation set accuracy: {}".format(np.mean(y_val_max == data["y_val"])))
print("Test set accuracy: {}".format(np.mean(y_test_max == data["y_test"])))

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

```
(Iteration 1 / 475) loss: 2.757300
(Epoch 0 / 5) train acc: 0.143000; val_acc: 0.174000
(Iteration 21 / 475) loss: 1.832566
(Iteration 41 / 475) loss: 1.513181
(Iteration 61 / 475) loss: 1.378851
(Iteration 81 / 475) loss: 1.234049
(Epoch 1 / 5) train acc: 0.602000; val_acc: 0.576000
(Iteration 101 / 475) loss: 1.286449
(Iteration 121 / 475) loss: 1.138128
(Iteration 141 / 475) loss: 1.164265
(Iteration 161 / 475) loss: 1.091654
(Iteration 181 / 475) loss: 1.038761
(Epoch 2 / 5) train acc: 0.654000; val_acc: 0.609000
(Iteration 201 / 475) loss: 1.060784
(Iteration 221 / 475) loss: 1.020408
(Iteration 241 / 475) loss: 0.995669
(Iteration 261 / 475) loss: 0.990324
(Iteration 281 / 475) loss: 0.985784
(Epoch 3 / 5) train acc: 0.762000; val_acc: 0.636000
(Iteration 301 / 475) loss: 0.954545
(Iteration 321 / 475) loss: 0.904532
(Iteration 341 / 475) loss: 0.936867
(Iteration 361 / 475) loss: 0.897486
(Epoch 4 / 5) train acc: 0.768000; val_acc: 0.647000
(Iteration 381 / 475) loss: 0.875304
(Iteration 401 / 475) loss: 0.790025
(Iteration 421 / 475) loss: 0.837693
(Iteration 441 / 475) loss: 0.829095
(Iteration 461 / 475) loss: 0.803787
(Epoch 5 / 5) train acc: 0.790000; val_acc: 0.667000
Validation set accuracy: 0.667
Test set accuracy: 0.654
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```
#cnn.py
import numpy as np

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

import pdb

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


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

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

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


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

    Inputs:
    - input_dim: Tuple (C, H, W) giving size of input data
```

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


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

    C, H, W = input_dim

    # CNN layer
    stride = 1
    pad = (filter_size - 1) / 2
    self.params["W1"] = np.random.normal(
        0, weight_scale, [num_filters, C, filter_size, filter_size]
    )
    self.params["b1"] = np.zeros([num_filters])

    # FC1
    h_out_cnn = (H - filter_size + 2 * pad) / stride + 1
    w_out_cnn = (W - filter_size + 2 * pad) / stride + 1
    h_out_pooling = int((h_out_cnn - 2) / 2 + 1)
    w_out_pooling = int((w_out_cnn - 2) / 2 + 1)
    self.params["W2"] = np.random.normal(
        0, weight_scale, [h_out_pooling * w_out_pooling * num_filters, hidden_dim]
```

```
    )
    self.params["b2"] = np.zeros([hidden_dim])

    # FC2
    self.params["W3"] = np.random.normal(0, weight_scale, [hidden_dim, num_classes])
    self.params["b3"] = np.zeros([num_classes])

    # batch norm layers
    if self.use_batchnorm:
        self.bn_params = []
        # CNN
        self.params["gamma1"] = np.ones(num_filters)
        self.params["beta1"] = np.zeros(num_filters)
        self.bn_params.append({"mode": "train"})
        # FC1
        self.params["gamma2"] = np.ones(hidden_dim)
        self.params["beta2"] = np.zeros(hidden_dim)
        self.bn_params.append({"mode": "train"})


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

        if self.use_batchnorm:
            # set mode
            mode = "test" if y is None else "train"
            for bn_param in self.bn_params:
                bn_param["mode"] = mode

            # get parameters
            gamma1, gamma2 = self.params["gamma1"], self.params["gamma2"]
            beta1, beta2 = self.params["beta1"], self.params["beta2"]
            bn_param1, bn_param2 = self.bn_params

            # foward CNN and FC1 layers
            out, cnn_cache = conv_bn_relu_pool_forward(
                X, W1, b1, conv_param, gamma1, beta1, bn_param1, pool_param
            )
            out, fc1_cache = affine_bn_relu_forward(out, W2, b2, gamma2, beta2, bn_param2)
        else:
            out, cnn_cache = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
            out, fc1_cache = affine_relu_forward(out, W2, b2)

        scores, fc2_cache = affine_forward(out, W3, b3)

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

        if y is None:
          return scores
```

```
        loss, grads = 0, {}
        # ================================================================ #
        # YOUR CODE HERE:
        #   Implement the backward pass of the three layer CNN.  Store the grads
        #   in the grads dictionary, exactly as before (i.e., the gradient of
        #   self.params[k] will be grads[k]).  Store the loss as "loss", and
        #   don't forget to add regularization on ALL weight matrices.
        # ================================================================ #

        # compute loss
        loss, dout = softmax_loss(scores, y)

        # add regularization loss
        for i in range(3):
            W = self.params["W" + str(i + 1)]
            loss += 0.5 * self.reg * (W * W).sum()

        # compute gradients
        dout, dw3, db3 = affine_backward(dout, fc2_cache)
        grads["W3"], grads["b3"] = dw3 + self.reg * W3, db3

        if self.use_batchnorm:
            dout, dw2, db2, dgamma2, dbeta2 = affine_bn_relu_backward(dout, fc1_cache)
            _, dw1, db1, dgamma1, dbeta1 = conv_bn_relu_pool_backward(dout, cnn_cache)
            grads["gamma1"], grads["gamma2"] = dgamma1, dgamma2
            grads["beta1"], grads["beta2"] = dbeta1, dbeta2
        else:
            dout, dw2, db2 = affine_relu_backward(dout, fc1_cache)
            _, dw1, db1 = conv_relu_pool_backward(dout, cnn_cache)

        grads["W2"], grads["b2"] = dw2 + self.reg * W2, db2
        grads["W1"], grads["b1"] = dw1 + self.reg * W1, db1

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

        return loss, grads
```

```
    pass
```

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



def conv_relu_forward(x, w, b, conv_param):
  """
  A convenience layer that performs a convolution followed by a ReLU.

  Inputs:
  - x: Input to the convolutional layer
  - w, b, conv_param: Weights and parameters for the convolutional layer

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


def conv_relu_backward(dout, cache):
  """
  Backward pass for the conv-relu convenience layer.
  """
  conv_cache, relu_cache = cache
```

```
  da = relu_backward(dout, relu_cache)
  dx, dw, db = conv_backward_fast(da, conv_cache)
  return dx, dw, db


def conv_relu_pool_forward(x, w, b, conv_param, pool_param):
  """
  Convenience layer that performs a convolution, a ReLU, and a pool.

  Inputs:
  - x: Input to the convolutional layer
  - w, b, conv_param: Weights and parameters for the convolutional layer
  - pool_param: Parameters for the pooling layer

  Returns a tuple of:
  - out: Output from the pooling layer
  - cache: Object to give to the backward pass
  """
  a, conv_cache = conv_forward_fast(x, w, b, conv_param)
  s, relu_cache = relu_forward(a)
  out, pool_cache = max_pool_forward_fast(s, pool_param)
  cache = (conv_cache, relu_cache, pool_cache)
  return out, cache


def conv_relu_pool_backward(dout, cache):
  """
  Backward pass for the conv-relu-pool convenience layer
  """
  conv_cache, relu_cache, pool_cache = cache
  ds = max_pool_backward_fast(dout, pool_cache)
  da = relu_backward(ds, relu_cache)
  dx, dw, db = conv_backward_fast(da, conv_cache)
  return dx, dw, db

def conv_bn_relu_pool_forward(x, w, b, conv_param, gamma, beta, bn_param, pool_param):
    """
    Convenience layer that performs a convolution, BN, a ReLU, and a pool.

    Inputs:
    - x: Input to the convolutional layer
```

```
    - w, b, conv_param: Weights and parameters for the convolutional layer
    - pool_param: Parameters for the pooling layer

    Returns a tuple of:
    - out: Output from the pooling layer
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    a_bn, bn_cache = spatial_batchnorm_forward(a, gamma, beta, bn_param)
    s, relu_cache = relu_forward(a_bn)
    out, pool_cache = max_pool_forward_fast(s, pool_param)
    cache = (conv_cache, bn_cache, relu_cache, pool_cache)
    return out, cache


def conv_bn_relu_pool_backward(dout, cache):
    """
    Backward pass for the conv-bn-relu-pool convenience layer
    """
    conv_cache, bn_cache, relu_cache, pool_cache = cache
    ds = max_pool_backward_fast(dout, pool_cache)
    da_bn = relu_backward(ds, relu_cache)
    da, dgamma, dbeta = spatial_batchnorm_backward(da_bn, bn_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db, dgamma, dbeta
```

Spatial batch normalization

In fully connected networks, we performed batch normalization on the activations. To do something equivalent on CNNs, we modify batch normalization slightly.

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization accepts inputs of shape `(N, C, H, W)` and produces outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

How do we calculate the spatial averages? First, notice that for the `C` feature maps we have (i.e., the layer has `C` filters) that each of these ought to have its own batch norm statistics, since each feature map may be picking out very different features in the images. However, within a feature map, we may assume that across all inputs and across all locations in the feature map, there ought to be relatively similar first and second order statistics. Hence, one way to think of spatial batch-normalization is to reshape the `(N, C, H, W)` array as an `(N*H*W, C)` array and perform batch normalization on this array.

Since spatial batch norm and batch normalization are similar, it'd be good to at this point also copy and paste our prior implemented layers from HW #4. Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

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

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

If you use your prior implementations of the batchnorm, then your spatial batchnorm implementation may be very short. Our implementations of the forward and backward pass are each 6 lines of code.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [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))))
```

Spatial batch normalization forward pass

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

In [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:  [ 9.67255611 10.01497911  9.38315379]
  Stds:  [3.78525534 4.66244975 4.03932632]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [-6.30051566e-16  2.60902411e-16  7.63278329e-17]
  Stds:  [0.99999965 0.99999977 0.99999969]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999895 3.99999908 4.99999847]
```

Spatial batch normalization backward pass

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

In [4]:
```
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:  1.4498310433403153e-08
dgamma error:  2.0937713372251024e-11
dbeta error:  3.2754748472125313e-12
```

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

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

```
cs231n.stanford.edu.
"""

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

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

  xpad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode="constant")
```

```
    out_height = int(1 + (H + 2 * pad - HH) / stride)
    out_width = int(1 + (W + 2 * pad - WW) / stride)
    out = np.zeros([N, F, out_height, out_width])

    for i in range(N):
      for c_i in range(F):
          for h_i in range(out_height):
              for w_i in range(out_width):
                  out[i, c_i, h_i, w_i] = (w[c_i]* xpad[i,:,h_i * stride : h_i * stride + HH,w_i * stride : w_i * stride + WW,])

    # =============================================================== #
    # 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.
    # ================================================================ #

    # dx with padding
    dx = np.zeros_like(xpad)

    for i in range(N):
      for c_i in range(F):
          for h_i in range(out_height):
              for w_i in range(out_width):
                  dx[i,:,h_i * stride : h_i * stride + f_height,w_i * stride : w_i * stride + f_width,] += (dout[i, c_i, h_i, w_

    # adjust dx shape
    H, W = x.shape[-2:]
    dx = dx[:, :, pad : H + pad, pad : W + pad]

    # dw
    dw = np.zeros_like(w)

    for i in range(N):
      for c_i in range(F):
          for h_i in range(out_height):
              for w_i in range(out_width):
                  dw[c_i] += (dout[i, c_i, h_i, w_i]* xpad[i,:,h_i * stride : h_i * stride + f_height,w_i * stride : w_i * strid

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

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

    return dx, dw, db


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

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

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

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

  pool_height = pool_param["pool_height"]
  pool_width = pool_param["pool_width"]
  stride = pool_param["stride"]

  N, C, H, W = x.shape
  out_height = int((H - pool_height) / stride + 1)
  out_width = int((W - pool_width) / stride + 1)
  out = np.zeros([N, C, out_height, out_width])

  for i in range(N):
    for c_i in range(C):
        for h_i in range(out_height):
            for w_i in range(out_width):
                out[i, c_i, h_i, w_i] = (x[i,c_i,h_i * stride : h_i * stride + pool_height,w_i * stride : w_i * stride + pool_

  # ================================================================ #
  # 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 = x.shape[:2]
  out_height, out_width = dout.shape[-2:]
  dx = np.zeros_like(x)

  for i in range(N):
    for c_i in range(C):
        for h_i in range(out_height):
            for w_i in range(out_width):
                max_idx_1d = np.argmax(x[i,c_i,h_i * stride : h_i * stride + pool_height,w_i * stride : w_i * stride + pool_wi
                max_idx_2d = np.unravel_index(max_idx_1d, [pool_height, pool_width])
                dx[i,c_i,h_i * stride + max_idx_2d[0],w_i * stride + max_idx_2d[1],] = dout[i, c_i, h_i, w_i]

  # ================================================================ #
  # 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_flatten = x.transpose(0, 2, 3, 1).reshape((N * H * W, C))
  out, cache = batchnorm_forward(x_flatten, gamma, beta, bn_param)
  out = out.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

  return out, cache


def spatial_batchnorm_backward(dout, cache):
  """
```

```
    Computes the backward pass for spatial batch normalization.

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

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

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

    N, C, H, W = dout.shape
    dout_flatten = dout.transpose((0, 2, 3, 1)).reshape((N * H * W, C))
    dx, dgamma, dbeta = batchnorm_backward(dout_flatten, cache)
    dx = dx.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

    return dx, dgamma, dbeta
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Convolutional neural network layers

In this notebook, we will build the convolutional neural network layers. This will be followed by a spatial batchnorm, and then in the final notebook of this assignment, we will train a CNN to further improve the validation accuracy on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [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))))
```

Implementing CNN layers

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

Convolutional forward pass

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

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

In [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
```

Convolutional backward pass

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

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

In [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:  3.433525827262979e-09
dw error:  2.0451881849575468e-10
db error:  9.314615301545273e-12
```

Max pool forward pass

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

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

In [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
```

Max pool backward pass

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

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

In [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:  3.2756373798792614e-12
```

Fast implementation of the CNN layers

Implementing fast versions of the CNN layers can be difficult. We will provide you with the fast layers implemented by cs231n. They are provided in `cs231n/fast_layers.py`.

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

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

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the cell below.

You should see pretty drastic speedups in the implementation of these layers. On our machine, the forward pass speeds up by 17x and the backward pass speeds up by 840x. Of course, these numbers will vary from machine to machine, as well as on your precise implementation of the naive layers.

In [6]:
```
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: 7.137939s
Fast: 0.011487s
Speedup: 621.392424x
Difference:  1.6670950699921258e-11

Testing conv_backward_fast:
Naive: 13.699482s
Fast: 0.025955s
Speedup: 527.807780x
dx difference:  1.4526197890903533e-11
dw difference:  9.944272022165403e-13
db difference:  0.0
```

In [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: 0.484370s
fast: 0.007977s
speedup: 60.718910x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.171220s
speedup: 58.574302x
dx difference:  0.0
```

Implementation of cascaded layers

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

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

In [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:  1.461256612960645e-08
dw error:  1.8088287106082316e-09
db error:  1.6615038614899664e-11
```

In [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:  2.4197688828352072e-09
dw error:  3.973827880551367e-10
db error:  2.539029185583785e-12
```

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.

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

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


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

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

  xpad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode="constant")
  out_height = int(1 + (H + 2 * pad - HH) / stride)
  out_width = int(1 + (W + 2 * pad - WW) / stride)
  out = np.zeros([N, F, out_height, out_width])

  for i in range(N):
    for c_i in range(F):
        for h_i in range(out_height):
            for w_i in range(out_width):
                out[i, c_i, h_i, w_i] = (w[c_i]* xpad[i,:,h_i * stride : h_i * stride + HH,w_i * stride : w_i * stride + WW,])

  # ================================================================ #
  # 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.
    # ================================================================ #

    # dx with padding
    dx = np.zeros_like(xpad)

    for i in range(N):
      for c_i in range(F):
          for h_i in range(out_height):
              for w_i in range(out_width):
                  dx[i,:,h_i * stride : h_i * stride + f_height,w_i * stride : w_i * stride + f_width,] += (dout[i, c_i, h_i, w_

    # adjust dx shape
    H, W = x.shape[-2:]
    dx = dx[:, :, pad : H + pad, pad : W + pad]

    # dw
    dw = np.zeros_like(w)

    for i in range(N):
      for c_i in range(F):
          for h_i in range(out_height):
              for w_i in range(out_width):
                  dw[c_i] += (dout[i, c_i, h_i, w_i]* xpad[i,:,h_i * stride : h_i * stride + f_height,w_i * stride : w_i * strid

    # db
```

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

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

    return dx, dw, db


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

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

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

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

    pool_height = pool_param["pool_height"]
    pool_width = pool_param["pool_width"]
    stride = pool_param["stride"]

    N, C, H, W = x.shape
    out_height = int((H - pool_height) / stride + 1)
    out_width = int((W - pool_width) / stride + 1)
    out = np.zeros([N, C, out_height, out_width])

    for i in range(N):
```

```
    for c_i in range(C):
        for h_i in range(out_height):
            for w_i in range(out_width):
                out[i, c_i, h_i, w_i] = (x[i,c_i,h_i * stride : h_i * stride + pool_height,w_i * stride : w_i * stride + pool_

  # ================================================================ #
  # 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 = x.shape[:2]
  out_height, out_width = dout.shape[-2:]
  dx = np.zeros_like(x)

  for i in range(N):
    for c_i in range(C):
        for h_i in range(out_height):
            for w_i in range(out_width):
                max_idx_1d = np.argmax(x[i,c_i,h_i * stride : h_i * stride + pool_height,w_i * stride : w_i * stride + pool_wi
                max_idx_2d = np.unravel_index(max_idx_1d, [pool_height, pool_width])
```

```
                dx[i,c_i,h_i * stride + max_idx_2d[0],w_i * stride + max_idx_2d[1],] = dout[i, c_i, h_i, w_i]

    # ================================================================ #
    # 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_flatten = x.transpose(0, 2, 3, 1).reshape((N * H * W, C))
    out, cache = batchnorm_forward(x_flatten, gamma, beta, bn_param)
    out = out.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

    return out, cache


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

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

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

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

    N, C, H, W = dout.shape
    dout_flatten = dout.transpose((0, 2, 3, 1)).reshape((N * H * W, C))
    dx, dgamma, dbeta = batchnorm_backward(dout_flatten, cache)
    dx = dx.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

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

    return dx, dgamma, dbeta
```