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 cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval
_numerical_gradient
from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-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_val: (1000, 3, 32, 32)
X_train: (49000, 3, 32, 32)
X_test: (1000, 3, 32, 32)
y_val: (1000,)
y_train: (49000,)
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 [8]:
```
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_na
me], verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('{} max relative error: {}'.format(param_name, rel_error(pa
ram_grad_num, grads[param_name])))
```

```
W1 max relative error: 0.00292987249321
W2 max relative error: 0.0155920132487
W3 max relative error: 7.11899484973e-05
b1 max relative error: 5.0406340691e-05
b2 max relative error: 1.92297083123e-05
b3 max relative error: 1.33771926975e-09
```

Overfit small dataset

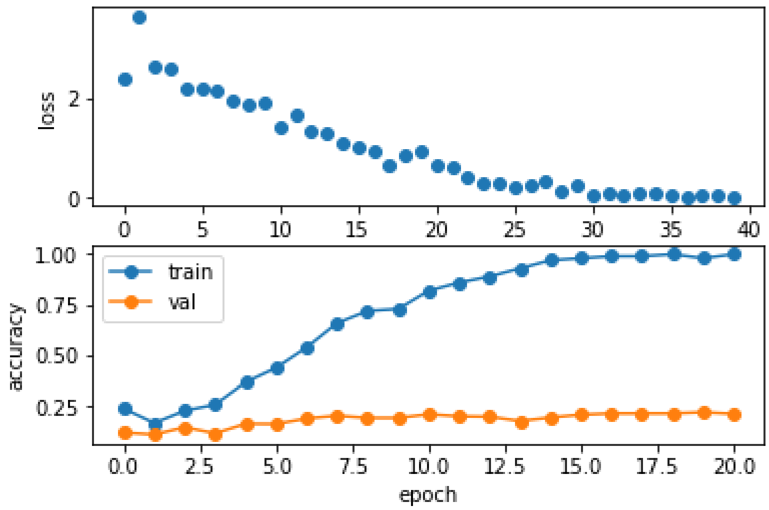
To check your CNN implementation, let's overfit a small dataset.

In [9]:
```
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=20, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 40) loss: 2.398724
(Epoch 0 / 20) train acc: 0.240000; val_acc: 0.123000
(Iteration 2 / 40) loss: 3.658866
(Epoch 1 / 20) train acc: 0.170000; val_acc: 0.113000
(Iteration 3 / 40) loss: 2.623158
(Iteration 4 / 40) loss: 2.581057
(Epoch 2 / 20) train acc: 0.230000; val_acc: 0.149000
(Iteration 5 / 40) loss: 2.198190
(Iteration 6 / 40) loss: 2.208519
(Epoch 3 / 20) train acc: 0.260000; val_acc: 0.119000
(Iteration 7 / 40) loss: 2.139135
(Iteration 8 / 40) loss: 1.961251
(Epoch 4 / 20) train acc: 0.370000; val_acc: 0.166000
(Iteration 9 / 40) loss: 1.886949
(Iteration 10 / 40) loss: 1.919783
(Epoch 5 / 20) train acc: 0.440000; val_acc: 0.165000
(Iteration 11 / 40) loss: 1.434302
(Iteration 12 / 40) loss: 1.668035
(Epoch 6 / 20) train acc: 0.540000; val_acc: 0.192000
(Iteration 13 / 40) loss: 1.352097
(Iteration 14 / 40) loss: 1.305671
(Epoch 7 / 20) train acc: 0.660000; val_acc: 0.206000
(Iteration 15 / 40) loss: 1.089647
(Iteration 16 / 40) loss: 1.030339
(Epoch 8 / 20) train acc: 0.720000; val_acc: 0.195000
(Iteration 17 / 40) loss: 0.933218
(Iteration 18 / 40) loss: 0.653353
(Epoch 9 / 20) train acc: 0.730000; val_acc: 0.195000
(Iteration 19 / 40) loss: 0.860156
(Iteration 20 / 40) loss: 0.956476
(Epoch 10 / 20) train acc: 0.820000; val_acc: 0.213000
(Iteration 21 / 40) loss: 0.673383
(Iteration 22 / 40) loss: 0.626792
(Epoch 11 / 20) train acc: 0.860000; val_acc: 0.203000
(Iteration 23 / 40) loss: 0.410349
(Iteration 24 / 40) loss: 0.297234
(Epoch 12 / 20) train acc: 0.890000; val_acc: 0.201000
(Iteration 25 / 40) loss: 0.280284
(Iteration 26 / 40) loss: 0.206168
(Epoch 13 / 20) train acc: 0.930000; val_acc: 0.180000
(Iteration 27 / 40) loss: 0.242705
(Iteration 28 / 40) loss: 0.332490
(Epoch 14 / 20) train acc: 0.970000; val_acc: 0.197000
(Iteration 29 / 40) loss: 0.138159
(Iteration 30 / 40) loss: 0.265789
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.211000
(Iteration 31 / 40) loss: 0.064454
(Iteration 32 / 40) loss: 0.102391
(Epoch 16 / 20) train acc: 0.990000; val_acc: 0.217000
(Iteration 33 / 40) loss: 0.049662
(Iteration 34 / 40) loss: 0.098386
(Epoch 17 / 20) train acc: 0.990000; val_acc: 0.217000
(Iteration 35 / 40) loss: 0.080395
(Iteration 36 / 40) loss: 0.069240
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.217000
(Iteration 37 / 40) loss: 0.026617
(Iteration 38 / 40) loss: 0.049096
```

```
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.223000
(Iteration 39 / 40) loss: 0.038309
(Iteration 40 / 40) loss: 0.021915
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.215000
```

In [10]:
```
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()
```



Train the network

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

In [11]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.0
01)

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: 2.304511
(Epoch 0 / 1) train acc: 0.128000; val_acc: 0.139000
(Iteration 21 / 980) loss: 2.231854
(Iteration 41 / 980) loss: 2.148167
(Iteration 61 / 980) loss: 1.939863
(Iteration 81 / 980) loss: 1.755125
(Iteration 101 / 980) loss: 1.797583
(Iteration 121 / 980) loss: 1.717661
(Iteration 141 / 980) loss: 1.942601
(Iteration 161 / 980) loss: 1.737939
(Iteration 181 / 980) loss: 1.765807
(Iteration 201 / 980) loss: 1.970439
(Iteration 221 / 980) loss: 1.943240
(Iteration 241 / 980) loss: 1.789837
(Iteration 261 / 980) loss: 1.686366
(Iteration 281 / 980) loss: 1.725475
(Iteration 301 / 980) loss: 1.504726
(Iteration 321 / 980) loss: 1.849093
(Iteration 341 / 980) loss: 1.331362
(Iteration 361 / 980) loss: 2.033096
(Iteration 381 / 980) loss: 1.936199
(Iteration 401 / 980) loss: 1.587918
(Iteration 421 / 980) loss: 1.521933
(Iteration 441 / 980) loss: 1.531843
(Iteration 461 / 980) loss: 1.640172
(Iteration 481 / 980) loss: 1.494496
(Iteration 501 / 980) loss: 1.694095
(Iteration 521 / 980) loss: 1.676323
(Iteration 541 / 980) loss: 1.590233
(Iteration 561 / 980) loss: 1.577411
(Iteration 581 / 980) loss: 1.630510
(Iteration 601 / 980) loss: 1.802195
(Iteration 621 / 980) loss: 1.790504
(Iteration 641 / 980) loss: 1.791112
(Iteration 661 / 980) loss: 1.720310
(Iteration 681 / 980) loss: 1.328894
(Iteration 701 / 980) loss: 1.714998
(Iteration 721 / 980) loss: 1.439794
(Iteration 741 / 980) loss: 1.322611
(Iteration 761 / 980) loss: 1.479585
(Iteration 781 / 980) loss: 1.402265
(Iteration 801 / 980) loss: 1.576688
(Iteration 821 / 980) loss: 1.518877
(Iteration 841 / 980) loss: 1.505881
(Iteration 861 / 980) loss: 1.708602
(Iteration 881 / 980) loss: 1.465694
(Iteration 901 / 980) loss: 1.518513
(Iteration 921 / 980) loss: 1.471163
(Iteration 941 / 980) loss: 1.342395
(Iteration 961 / 980) loss: 1.610166
(Epoch 1 / 1) train acc: 0.494000; val_acc: 0.472000
```

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 [3]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.0
01,
                         num_filters=32)

solver = Solver(model, data,
                num_epochs=20, batch_size=200,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)

solver.train()

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

```
(Iteration 1 / 4900) loss: 2.304641
(Epoch 0 / 20) train acc: 0.099000; val_acc: 0.087000
(Iteration 21 / 4900) loss: 2.022833
(Iteration 41 / 4900) loss: 1.829609
(Iteration 61 / 4900) loss: 1.578786
(Iteration 81 / 4900) loss: 1.525048
(Iteration 101 / 4900) loss: 1.524534
(Iteration 121 / 4900) loss: 1.491584
(Iteration 141 / 4900) loss: 1.484886
(Iteration 161 / 4900) loss: 1.451082
(Iteration 181 / 4900) loss: 1.384444
(Iteration 201 / 4900) loss: 1.391388
(Iteration 221 / 4900) loss: 1.263857
(Iteration 241 / 4900) loss: 1.477539
(Epoch 1 / 20) train acc: 0.507000; val_acc: 0.493000
(Iteration 261 / 4900) loss: 1.262600
(Iteration 281 / 4900) loss: 1.369997
(Iteration 301 / 4900) loss: 1.176921
(Iteration 321 / 4900) loss: 1.533619
(Iteration 341 / 4900) loss: 1.322418
(Iteration 361 / 4900) loss: 1.153445
(Iteration 381 / 4900) loss: 1.329984
(Iteration 401 / 4900) loss: 1.336941
(Iteration 421 / 4900) loss: 1.231730
(Iteration 441 / 4900) loss: 1.078800
(Iteration 461 / 4900) loss: 1.286022
(Iteration 481 / 4900) loss: 1.337707
(Epoch 2 / 20) train acc: 0.605000; val_acc: 0.584000
(Iteration 501 / 4900) loss: 1.134415
(Iteration 521 / 4900) loss: 1.144201
(Iteration 541 / 4900) loss: 1.010227
(Iteration 561 / 4900) loss: 1.293241
(Iteration 581 / 4900) loss: 1.193454
(Iteration 601 / 4900) loss: 1.102433
(Iteration 621 / 4900) loss: 1.153127
(Iteration 641 / 4900) loss: 0.996380
(Iteration 661 / 4900) loss: 1.083883
(Iteration 681 / 4900) loss: 1.194233
(Iteration 701 / 4900) loss: 1.146079
(Iteration 721 / 4900) loss: 1.307219
(Epoch 3 / 20) train acc: 0.626000; val_acc: 0.603000
(Iteration 741 / 4900) loss: 1.176179
(Iteration 761 / 4900) loss: 1.221810
(Iteration 781 / 4900) loss: 1.322550
(Iteration 801 / 4900) loss: 1.156965
(Iteration 821 / 4900) loss: 1.254104
(Iteration 841 / 4900) loss: 0.918489
(Iteration 861 / 4900) loss: 1.142895
(Iteration 881 / 4900) loss: 1.190759
(Iteration 901 / 4900) loss: 1.099620
(Iteration 921 / 4900) loss: 1.129262
(Iteration 941 / 4900) loss: 1.066536
(Iteration 961 / 4900) loss: 1.286184
(Epoch 4 / 20) train acc: 0.636000; val_acc: 0.624000
(Iteration 981 / 4900) loss: 1.030652
(Iteration 1001 / 4900) loss: 1.120823
(Iteration 1021 / 4900) loss: 1.030835
```

```
(Iteration 1041 / 4900) loss: 1.124149
(Iteration 1061 / 4900) loss: 1.102276
(Iteration 1081 / 4900) loss: 0.881765
(Iteration 1101 / 4900) loss: 1.156674
(Iteration 1121 / 4900) loss: 1.233775
(Iteration 1141 / 4900) loss: 1.089831
(Iteration 1161 / 4900) loss: 0.990193
(Iteration 1181 / 4900) loss: 1.056324
(Iteration 1201 / 4900) loss: 1.150618
(Iteration 1221 / 4900) loss: 0.922874
(Epoch 5 / 20) train acc: 0.708000; val_acc: 0.631000
(Iteration 1241 / 4900) loss: 0.953913
(Iteration 1261 / 4900) loss: 0.953533
(Iteration 1281 / 4900) loss: 1.131170
(Iteration 1301 / 4900) loss: 1.108464
(Iteration 1321 / 4900) loss: 0.915440
(Iteration 1341 / 4900) loss: 0.964860
(Iteration 1361 / 4900) loss: 1.069142
(Iteration 1381 / 4900) loss: 1.014813
(Iteration 1401 / 4900) loss: 1.015884
(Iteration 1421 / 4900) loss: 1.091674
(Iteration 1441 / 4900) loss: 0.916118
(Iteration 1461 / 4900) loss: 0.929332
(Epoch 6 / 20) train acc: 0.691000; val_acc: 0.624000
(Iteration 1481 / 4900) loss: 0.874445
(Iteration 1501 / 4900) loss: 0.853501
(Iteration 1521 / 4900) loss: 0.960778
(Iteration 1541 / 4900) loss: 0.990698
(Iteration 1561 / 4900) loss: 1.192997
(Iteration 1581 / 4900) loss: 1.125338
(Iteration 1601 / 4900) loss: 0.965832
(Iteration 1621 / 4900) loss: 1.030510
(Iteration 1641 / 4900) loss: 1.083969
(Iteration 1661 / 4900) loss: 0.859961
(Iteration 1681 / 4900) loss: 0.886632
(Iteration 1701 / 4900) loss: 0.962145
(Epoch 7 / 20) train acc: 0.691000; val_acc: 0.597000
(Iteration 1721 / 4900) loss: 0.983099
(Iteration 1741 / 4900) loss: 0.940587
(Iteration 1761 / 4900) loss: 1.006831
(Iteration 1781 / 4900) loss: 0.881595
(Iteration 1801 / 4900) loss: 0.949778
(Iteration 1821 / 4900) loss: 0.872129
(Iteration 1841 / 4900) loss: 0.973894
(Iteration 1861 / 4900) loss: 0.881316
(Iteration 1881 / 4900) loss: 0.952845
(Iteration 1901 / 4900) loss: 0.882571
(Iteration 1921 / 4900) loss: 0.979896
(Iteration 1941 / 4900) loss: 0.956969
(Epoch 8 / 20) train acc: 0.717000; val_acc: 0.652000
(Iteration 1961 / 4900) loss: 0.892523
(Iteration 1981 / 4900) loss: 0.834394
(Iteration 2001 / 4900) loss: 0.917589
```

```
-------------------------------------------------------------------------
--
KeyboardInterrupt                         Traceback (most recent call
 last)
<ipython-input-3-1e80315e6404> in <module>()
     15                  verbose=True, print_every=20)
     16
---> 17 solver.train()
     18
     19 # ===========================================================
===== #

/home/ben/Documents/239AS/HW5/code/cs231n/solver.pyc in train(self)
    262
    263         for t in range(num_iterations):
--> 264             self._step()
    265
    266             # Maybe print training loss

/home/ben/Documents/239AS/HW5/code/cs231n/solver.pyc in _step(self)
    178
    179         # Compute loss and gradient
--> 180         loss, grads = self.model.loss(X_batch, y_batch)
    181         self.loss_history.append(loss)
    182

/home/ben/Documents/239AS/HW5/code/nndl/cnn.pyc in loss(self, X, y)
    102     # ======================================================
========= #
    103
--> 104     c1, conv_cache1 = conv_forward_fast(X, W1, b1,
conv_param)
    105     h1, r_cache1 = relu_forward(c1)
    106     mp1, mp_cache1 = max_pool_forward_fast(h1, pool_param)

/home/ben/Documents/239AS/HW5/code/cs231n/fast_layers.pyc in conv_for
ward_strides(x, w, b, conv_param)
     72
     73     # Now all our convolutions are a big matrix multiply
---> 74     res = w.reshape(F, -1).dot(x_cols) + b.reshape(-1, 1)
     75
     76     # Reshape the output

KeyboardInterrupt:
```

In [4]:
```
print solver.best_val_acc
```

```
0.652
```