CNN

February 25, 2020

1 Convolutional neural networks

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

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_gradie
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 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,)
```

1.1 Three layer CNN

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

The architecture of this CNN will be:

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

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

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

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=Fals
            e = rel_error(param_grad_num, grads[param_name])
            print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, gra
```

```
W1 max relative error: 0.0006507149025299291
W2 max relative error: 0.00568316345265749
W3 max relative error: 4.2889602827172216e-05
b1 max relative error: 1.7735505300977068e-05
b2 max relative error: 8.581484252488379e-07
b3 max relative error: 1.1508232107915781e-09
```

1.1.1 Overfit small dataset

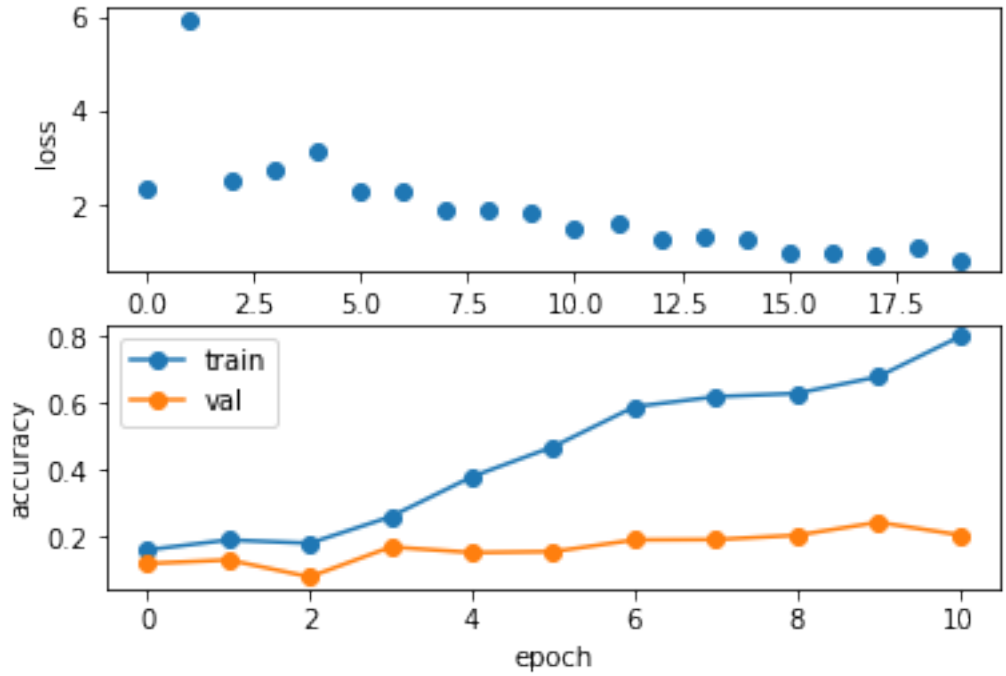
To check your CNN implementation, let's overfit a small dataset.

```
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: 2.381830
(Epoch 0 / 10) train acc: 0.160000; val_acc: 0.119000
(Iteration 2 / 20) loss: 5.928924
(Epoch 1 / 10) train acc: 0.190000; val_acc: 0.130000
(Iteration 3 / 20) loss: 2.551710
(Iteration 4 / 20) loss: 2.735981
(Epoch 2 / 10) train acc: 0.180000; val_acc: 0.079000
(Iteration 5 / 20) loss: 3.169556
(Iteration 6 / 20) loss: 2.329534
(Epoch 3 / 10) train acc: 0.260000; val_acc: 0.169000
```

```
(Iteration 7 / 20) loss: 2.298394
(Iteration 8 / 20) loss: 1.884549
(Epoch 4 / 10) train acc: 0.380000; val_acc: 0.152000
(Iteration 9 / 20) loss: 1.923281
(Iteration 10 / 20) loss: 1.841728
(Epoch 5 / 10) train acc: 0.470000; val_acc: 0.155000
(Iteration 11 / 20) loss: 1.494036
(Iteration 12 / 20) loss: 1.623882
(Epoch 6 / 10) train acc: 0.590000; val_acc: 0.190000
(Iteration 13 / 20) loss: 1.293669
(Iteration 14 / 20) loss: 1.343754
(Epoch 7 / 10) train acc: 0.620000; val_acc: 0.191000
(Iteration 15 / 20) loss: 1.286494
(Iteration 16 / 20) loss: 0.993775
(Epoch 8 / 10) train acc: 0.630000; val_acc: 0.204000
(Iteration 17 / 20) loss: 0.985476
(Iteration 18 / 20) loss: 0.925707
(Epoch 9 / 10) train acc: 0.680000; val_acc: 0.243000
(Iteration 19 / 20) loss: 1.135033
(Iteration 20 / 20) loss: 0.856049
(Epoch 10 / 10) train acc: 0.800000; val_acc: 0.205000
```

In [5]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

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



1.2 Train the network

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

```
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: 2.304654
(Epoch 0 / 1) train acc: 0.102000; val_acc: 0.093000
(Iteration 21 / 980) loss: 2.131816
(Iteration 41 / 980) loss: 2.020851
(Iteration 61 / 980) loss: 2.117923
(Iteration 81 / 980) loss: 2.010330
(Iteration 101 / 980) loss: 1.687490
(Iteration 121 / 980) loss: 1.882670
(Iteration 141 / 980) loss: 1.939549
(Iteration 161 / 980) loss: 1.552810
```

```
(Iteration 181 / 980) loss: 1.823595
(Iteration 201 / 980) loss: 1.808657
(Iteration 221 / 980) loss: 1.854325
(Iteration 241 / 980) loss: 1.761188
(Iteration 261 / 980) loss: 1.800233
(Iteration 281 / 980) loss: 1.567720
(Iteration 301 / 980) loss: 1.539409
(Iteration 321 / 980) loss: 2.138078
(Iteration 341 / 980) loss: 1.746798
(Iteration 361 / 980) loss: 1.840328
(Iteration 381 / 980) loss: 1.881169
(Iteration 401 / 980) loss: 1.591436
(Iteration 421 / 980) loss: 1.656724
(Iteration 441 / 980) loss: 1.778309
(Iteration 461 / 980) loss: 1.720966
(Iteration 481 / 980) loss: 1.856955
(Iteration 501 / 980) loss: 1.624138
(Iteration 521 / 980) loss: 1.587256
(Iteration 541 / 980) loss: 1.542011
(Iteration 561 / 980) loss: 1.555545
(Iteration 581 / 980) loss: 1.396170
(Iteration 601 / 980) loss: 1.293165
(Iteration 621 / 980) loss: 1.270821
(Iteration 641 / 980) loss: 1.691648
(Iteration 661 / 980) loss: 1.570260
(Iteration 681 / 980) loss: 1.743356
(Iteration 701 / 980) loss: 1.474436
(Iteration 721 / 980) loss: 1.414279
(Iteration 741 / 980) loss: 1.339923
(Iteration 761 / 980) loss: 1.782336
(Iteration 781 / 980) loss: 1.740092
(Iteration 801 / 980) loss: 1.427833
(Iteration 821 / 980) loss: 1.437471
(Iteration 841 / 980) loss: 1.542376
(Iteration 861 / 980) loss: 1.576817
(Iteration 881 / 980) loss: 1.577748
(Iteration 901 / 980) loss: 1.722746
(Iteration 921 / 980) loss: 1.507274
(Iteration 941 / 980) loss: 1.371046
(Iteration 961 / 980) loss: 1.376196
(Epoch 1 / 1) train acc: 0.428000; val_acc: 0.427000
```

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

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

2.0.1 Things you should try:

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

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

2.0.2 Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple of 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 [9]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #

optimizer = 'adam'

learning_rate = 1e-3
lr_decay = 0.9
num_filters = 64
filter_size = 7

model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=20, batch_size=100,
                update_rule=optimizer,
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=lr_decay,
```

```
                      verbose=True, print_every=50)
        solver.train()

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

```
(Iteration 1 / 9800) loss: 2.304452
(Epoch 0 / 20) train acc: 0.116000; val_acc: 0.105000
(Iteration 51 / 9800) loss: 1.853642
(Iteration 101 / 9800) loss: 1.754378
(Iteration 151 / 9800) loss: 1.585016
(Iteration 201 / 9800) loss: 1.665315
(Iteration 251 / 9800) loss: 1.494164
(Iteration 301 / 9800) loss: 1.452221
(Iteration 351 / 9800) loss: 1.353152
(Iteration 401 / 9800) loss: 1.504349
(Iteration 451 / 9800) loss: 1.365023
(Epoch 1 / 20) train acc: 0.466000; val_acc: 0.491000
(Iteration 501 / 9800) loss: 1.385573
(Iteration 551 / 9800) loss: 1.383519
(Iteration 601 / 9800) loss: 1.342313
(Iteration 651 / 9800) loss: 1.418668
(Iteration 701 / 9800) loss: 1.202109
(Iteration 751 / 9800) loss: 1.360571
(Iteration 801 / 9800) loss: 1.141037
(Iteration 851 / 9800) loss: 1.203072
(Iteration 901 / 9800) loss: 1.451928
(Iteration 951 / 9800) loss: 1.349903
(Epoch 2 / 20) train acc: 0.530000; val_acc: 0.550000
(Iteration 1001 / 9800) loss: 1.270969
(Iteration 1051 / 9800) loss: 1.315878
(Iteration 1101 / 9800) loss: 1.212279
(Iteration 1151 / 9800) loss: 1.474329
(Iteration 1201 / 9800) loss: 1.144212
(Iteration 1251 / 9800) loss: 1.383498
(Iteration 1301 / 9800) loss: 1.388701
(Iteration 1351 / 9800) loss: 1.153182
(Iteration 1401 / 9800) loss: 1.103188
(Iteration 1451 / 9800) loss: 1.214557
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.578000
(Iteration 1501 / 9800) loss: 1.166137
(Iteration 1551 / 9800) loss: 1.163752
(Iteration 1601 / 9800) loss: 1.019426
(Iteration 1651 / 9800) loss: 1.259598
(Iteration 1701 / 9800) loss: 1.119750
(Iteration 1751 / 9800) loss: 1.221735
(Iteration 1801 / 9800) loss: 1.102449
```

```
(Iteration 1851 / 9800) loss: 1.122105
(Iteration 1901 / 9800) loss: 1.149631
(Iteration 1951 / 9800) loss: 1.077087
(Epoch 4 / 20) train acc: 0.650000; val_acc: 0.619000
(Iteration 2001 / 9800) loss: 0.993838
(Iteration 2051 / 9800) loss: 1.002742
(Iteration 2101 / 9800) loss: 0.910465
(Iteration 2151 / 9800) loss: 1.000292
(Iteration 2201 / 9800) loss: 0.944183
(Iteration 2251 / 9800) loss: 1.031122
(Iteration 2301 / 9800) loss: 0.908113
(Iteration 2351 / 9800) loss: 1.062253
(Iteration 2401 / 9800) loss: 1.203044
(Epoch 5 / 20) train acc: 0.664000; val_acc: 0.616000
(Iteration 2451 / 9800) loss: 1.003779
(Iteration 2501 / 9800) loss: 1.040453
(Iteration 2551 / 9800) loss: 0.933268
(Iteration 2601 / 9800) loss: 0.988178
(Iteration 2651 / 9800) loss: 1.014140
(Iteration 2701 / 9800) loss: 0.793451
(Iteration 2751 / 9800) loss: 1.217258
(Iteration 2801 / 9800) loss: 1.016398
(Iteration 2851 / 9800) loss: 0.805968
(Iteration 2901 / 9800) loss: 1.175808
(Epoch 6 / 20) train acc: 0.656000; val_acc: 0.605000
(Iteration 2951 / 9800) loss: 0.941986
(Iteration 3001 / 9800) loss: 0.982846
(Iteration 3051 / 9800) loss: 1.005875
(Iteration 3101 / 9800) loss: 0.929024
(Iteration 3151 / 9800) loss: 0.911446
(Iteration 3201 / 9800) loss: 0.831955
(Iteration 3251 / 9800) loss: 0.983365
(Iteration 3301 / 9800) loss: 1.084643
(Iteration 3351 / 9800) loss: 0.882134
(Iteration 3401 / 9800) loss: 0.907717
(Epoch 7 / 20) train acc: 0.693000; val_acc: 0.643000
(Iteration 3451 / 9800) loss: 0.926780
(Iteration 3501 / 9800) loss: 0.871370
(Iteration 3551 / 9800) loss: 0.916746
(Iteration 3601 / 9800) loss: 0.957859
(Iteration 3651 / 9800) loss: 0.945472
(Iteration 3701 / 9800) loss: 0.957690
(Iteration 3751 / 9800) loss: 0.890934
(Iteration 3801 / 9800) loss: 0.926908
(Iteration 3851 / 9800) loss: 1.042560
(Iteration 3901 / 9800) loss: 0.858834
(Epoch 8 / 20) train acc: 0.662000; val_acc: 0.633000
(Iteration 3951 / 9800) loss: 0.949428
```

```
(Iteration 4001 / 9800) loss: 1.022474
(Iteration 4051 / 9800) loss: 0.992603
(Iteration 4101 / 9800) loss: 1.047574
(Iteration 4151 / 9800) loss: 0.931832
(Iteration 4201 / 9800) loss: 0.772472
(Iteration 4251 / 9800) loss: 0.980237
(Iteration 4301 / 9800) loss: 1.027331
(Iteration 4351 / 9800) loss: 0.847622
(Iteration 4401 / 9800) loss: 0.800675
(Epoch 9 / 20) train acc: 0.737000; val_acc: 0.638000
(Iteration 4451 / 9800) loss: 0.834715
(Iteration 4501 / 9800) loss: 0.677432
(Iteration 4551 / 9800) loss: 1.080039
(Iteration 4601 / 9800) loss: 0.716728
(Iteration 4651 / 9800) loss: 0.806540
(Iteration 4701 / 9800) loss: 0.927516
(Iteration 4751 / 9800) loss: 0.875024
(Iteration 4801 / 9800) loss: 1.042306
(Iteration 4851 / 9800) loss: 0.882808
(Epoch 10 / 20) train acc: 0.721000; val_acc: 0.647000
(Iteration 4901 / 9800) loss: 0.892720
(Iteration 4951 / 9800) loss: 0.772937
(Iteration 5001 / 9800) loss: 0.810763
(Iteration 5051 / 9800) loss: 0.746560
(Iteration 5101 / 9800) loss: 0.696696
(Iteration 5151 / 9800) loss: 0.815378
(Iteration 5201 / 9800) loss: 0.859296
(Iteration 5251 / 9800) loss: 0.845369
(Iteration 5301 / 9800) loss: 0.803124
(Iteration 5351 / 9800) loss: 0.678393
(Epoch 11 / 20) train acc: 0.741000; val_acc: 0.662000
(Iteration 5401 / 9800) loss: 0.717994
(Iteration 5451 / 9800) loss: 0.730793
(Iteration 5501 / 9800) loss: 0.832582
(Iteration 5551 / 9800) loss: 0.773037
(Iteration 5601 / 9800) loss: 0.845416
(Iteration 5651 / 9800) loss: 0.783072
(Iteration 5701 / 9800) loss: 0.775187
(Iteration 5751 / 9800) loss: 0.882563
(Iteration 5801 / 9800) loss: 1.027854
(Iteration 5851 / 9800) loss: 0.830116
(Epoch 12 / 20) train acc: 0.759000; val_acc: 0.651000
(Iteration 5901 / 9800) loss: 0.705709
(Iteration 5951 / 9800) loss: 0.968117
(Iteration 6001 / 9800) loss: 0.736151
(Iteration 6051 / 9800) loss: 0.869076
(Iteration 6101 / 9800) loss: 0.766911
(Iteration 6151 / 9800) loss: 0.927024
```

```
(Iteration 6201 / 9800) loss: 0.662418
(Iteration 6251 / 9800) loss: 0.801865
(Iteration 6301 / 9800) loss: 0.654079
(Iteration 6351 / 9800) loss: 0.785943
(Epoch 13 / 20) train acc: 0.754000; val_acc: 0.648000
(Iteration 6401 / 9800) loss: 0.883470
(Iteration 6451 / 9800) loss: 0.720435
(Iteration 6501 / 9800) loss: 0.812503
(Iteration 6551 / 9800) loss: 0.775922
(Iteration 6601 / 9800) loss: 0.895231
(Iteration 6651 / 9800) loss: 0.766190
(Iteration 6701 / 9800) loss: 0.717640
(Iteration 6751 / 9800) loss: 0.738909
(Iteration 6801 / 9800) loss: 0.755856
(Iteration 6851 / 9800) loss: 0.754592
(Epoch 14 / 20) train acc: 0.751000; val_acc: 0.651000
(Iteration 6901 / 9800) loss: 0.691371
(Iteration 6951 / 9800) loss: 0.669393
(Iteration 7001 / 9800) loss: 0.748512
(Iteration 7051 / 9800) loss: 0.754870
(Iteration 7101 / 9800) loss: 0.611698
(Iteration 7151 / 9800) loss: 0.620703
(Iteration 7201 / 9800) loss: 0.804109
(Iteration 7251 / 9800) loss: 0.721399
(Iteration 7301 / 9800) loss: 0.729168
(Epoch 15 / 20) train acc: 0.797000; val_acc: 0.663000
(Iteration 7351 / 9800) loss: 0.794741
(Iteration 7401 / 9800) loss: 0.685225
(Iteration 7451 / 9800) loss: 0.659901
(Iteration 7501 / 9800) loss: 0.654125
(Iteration 7551 / 9800) loss: 0.897542
(Iteration 7601 / 9800) loss: 0.702215
(Iteration 7651 / 9800) loss: 0.958143
(Iteration 7701 / 9800) loss: 0.776002
(Iteration 7751 / 9800) loss: 0.690163
(Iteration 7801 / 9800) loss: 0.636967
(Epoch 16 / 20) train acc: 0.773000; val_acc: 0.651000
(Iteration 7851 / 9800) loss: 0.746292
(Iteration 7901 / 9800) loss: 0.577357
(Iteration 7951 / 9800) loss: 0.663366
(Iteration 8001 / 9800) loss: 0.851368
(Iteration 8051 / 9800) loss: 0.727580
(Iteration 8101 / 9800) loss: 0.683947
(Iteration 8151 / 9800) loss: 0.626854
(Iteration 8201 / 9800) loss: 0.800301
(Iteration 8251 / 9800) loss: 0.736672
(Iteration 8301 / 9800) loss: 0.818214
(Epoch 17 / 20) train acc: 0.760000; val_acc: 0.650000
```

```
(Iteration 8351 / 9800) loss: 0.802396
(Iteration 8401 / 9800) loss: 0.771348
(Iteration 8451 / 9800) loss: 0.622872
(Iteration 8501 / 9800) loss: 0.513887
(Iteration 8551 / 9800) loss: 0.539145
(Iteration 8601 / 9800) loss: 0.769144
(Iteration 8651 / 9800) loss: 0.708705
(Iteration 8701 / 9800) loss: 0.758026
(Iteration 8751 / 9800) loss: 0.806010
(Iteration 8801 / 9800) loss: 0.568705
(Epoch 18 / 20) train acc: 0.800000; val_acc: 0.661000
(Iteration 8851 / 9800) loss: 0.646179
(Iteration 8901 / 9800) loss: 0.531616
(Iteration 8951 / 9800) loss: 0.604449
(Iteration 9001 / 9800) loss: 0.779766
(Iteration 9051 / 9800) loss: 0.622247
(Iteration 9101 / 9800) loss: 0.680818
(Iteration 9151 / 9800) loss: 0.428375
(Iteration 9201 / 9800) loss: 0.928653
(Iteration 9251 / 9800) loss: 0.379808
(Iteration 9301 / 9800) loss: 0.789477
(Epoch 19 / 20) train acc: 0.817000; val_acc: 0.646000
(Iteration 9351 / 9800) loss: 0.763423
(Iteration 9401 / 9800) loss: 0.797257
(Iteration 9451 / 9800) loss: 0.708895
(Iteration 9501 / 9800) loss: 0.808784
(Iteration 9551 / 9800) loss: 0.568338
(Iteration 9601 / 9800) loss: 0.553148
(Iteration 9651 / 9800) loss: 0.637789
(Iteration 9701 / 9800) loss: 0.642199
(Iteration 9751 / 9800) loss: 0.614586
(Epoch 20 / 20) train acc: 0.807000; val_acc: 0.652000
```