Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive over 55% 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, and their layer structure. 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.fc_net import *
from nndl.layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradie
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-ipytho
%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,)
```

Dropout forward pass

Implement the training and test time dropout forward pass, `dropout_forward` , in `nndl/layers.py` . After that, test your implementation by running the following cell.

In [3]:
```
x = np.random.randn(500, 500) + 10

for p in [0.3, 0.6, 0.75]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
```

```
Running tests with p =  0.3
Mean of input:  9.998821555175194
Mean of train-time output:  9.998152942395887
Mean of test-time output:  9.998821555175194
Fraction of train-time output set to zero:  0.699916
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.998821555175194
Mean of train-time output:  9.99687326207575
Mean of test-time output:  9.998821555175194
Fraction of train-time output set to zero:  0.400044
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.998821555175194
Mean of train-time output:  10.014399455238639
Mean of test-time output:  9.998821555175194
Fraction of train-time output set to zero:  0.248896
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

Implement the backward pass, `dropout_backward` , in `nndl/layers.py` . After that, test your gradients by running the following cell:

In [4]:
```
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_pa

print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.445610702965805e-11
```

Implement a fully connected neural network with dropout layers

Modify the `FullyConnectedNet()` class in `nndl/fc_net.py` to incorporate dropout. A dropout layer should be incorporated after every ReLU layer. Concretely, there shouldn't be a dropout at the output layer since there is no ReLU at the output layer. You will need to modify the class in the following areas:

(1) In the forward pass, you will need to incorporate a dropout layer after every relu layer.

(2) In the backward pass, you will need to incorporate a dropout backward pass layer.

Check your implementation by running the following code. Our W1 gradient relative error is on the order of 1e-6 (the largest of all the relative errors).

In [5]:

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [0, 0.25, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[name])))
  print('\n')
```

```
Running check with dropout =  0
Initial loss:  2.3051948273987857
W1 relative error: 2.527257534437607e-07
W2 relative error: 1.5034484929313676e-05
W3 relative error: 2.753446833630168e-07
b1 relative error: 2.936957476400148e-06
b2 relative error: 5.051339805546953e-08
b3 relative error: 1.1740467838205477e-10


Running check with dropout =  0.25
Initial loss:  2.3126468345657742
W1 relative error: 1.483854795975875e-08
W2 relative error: 2.3427832149940254e-10
W3 relative error: 3.564454999162522e-08
b1 relative error: 1.5292167232408546e-09
b2 relative error: 1.842268684106 78e-10
b3 relative error: 8.701800136729388e-11


Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.553387957138422e-08
W2 relative error: 2.974218050584597e-08
W3 relative error: 4.3413247403122424e-07
b1 relative error: 1.872462967441693e-08
b2 relative error: 5.045591219274328e-09
b3 relative error: 7.487013797161614e-11
```

Dropout as a regularizer

In class, we claimed that dropout acts as a regularizer by effectively bagging. To check this, we will train two small networks, one with dropout and one without dropout.

In [6]:
```
# Train two identical nets, one with dropout and one without

num_train = 500
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'],
}

solvers = {}
dropout_choices = [0, 0.6]
for dropout in dropout_choices:
  model = FullyConnectedNet([100, 100, 100], dropout=dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

```
(Iteration 1 / 125) loss: 2.300804
(Epoch 0 / 25) train acc: 0.220000; val_acc: 0.168000
(Epoch 1 / 25) train acc: 0.188000; val_acc: 0.147000
(Epoch 2 / 25) train acc: 0.266000; val_acc: 0.200000
(Epoch 3 / 25) train acc: 0.338000; val_acc: 0.262000
(Epoch 4 / 25) train acc: 0.378000; val_acc: 0.278000
(Epoch 5 / 25) train acc: 0.428000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.468000; val_acc: 0.323000
(Epoch 7 / 25) train acc: 0.494000; val_acc: 0.287000
(Epoch 8 / 25) train acc: 0.566000; val_acc: 0.328000
(Epoch 9 / 25) train acc: 0.572000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.622000; val_acc: 0.324000
(Epoch 11 / 25) train acc: 0.670000; val_acc: 0.279000
(Epoch 12 / 25) train acc: 0.710000; val_acc: 0.338000
(Epoch 13 / 25) train acc: 0.746000; val_acc: 0.319000
(Epoch 14 / 25) train acc: 0.792000; val_acc: 0.307000
(Epoch 15 / 25) train acc: 0.834000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.876000; val_acc: 0.327000
(Epoch 17 / 25) train acc: 0.886000; val_acc: 0.320000
(Epoch 18 / 25) train acc: 0.918000; val_acc: 0.314000
(Epoch 19 / 25) train acc: 0.922000; val_acc: 0.290000
(Epoch 20 / 25) train acc: 0.944000; val_acc: 0.306000
(Iteration 101 / 125) loss: 0.156105
(Epoch 21 / 25) train acc: 0.968000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.302000
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.986000; val_acc: 0.285000
```

```
(Epoch 25 / 25) train acc: 0.978000; val_acc: 0.311000
(Iteration 1 / 125) loss: 2.301328
(Epoch 0 / 25) train acc: 0.154000; val_acc: 0.143000
(Epoch 1 / 25) train acc: 0.214000; val_acc: 0.195000
(Epoch 2 / 25) train acc: 0.252000; val_acc: 0.216000
(Epoch 3 / 25) train acc: 0.276000; val_acc: 0.200000
(Epoch 4 / 25) train acc: 0.308000; val_acc: 0.254000
(Epoch 5 / 25) train acc: 0.316000; val_acc: 0.241000
(Epoch 6 / 25) train acc: 0.322000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.354000; val_acc: 0.273000
(Epoch 8 / 25) train acc: 0.364000; val_acc: 0.276000
(Epoch 9 / 25) train acc: 0.408000; val_acc: 0.282000
(Epoch 10 / 25) train acc: 0.454000; val_acc: 0.302000
(Epoch 11 / 25) train acc: 0.472000; val_acc: 0.296000
(Epoch 12 / 25) train acc: 0.496000; val_acc: 0.318000
(Epoch 13 / 25) train acc: 0.512000; val_acc: 0.310000
(Epoch 14 / 25) train acc: 0.532000; val_acc: 0.318000
(Epoch 15 / 25) train acc: 0.558000; val_acc: 0.331000
(Epoch 16 / 25) train acc: 0.574000; val_acc: 0.300000
(Epoch 17 / 25) train acc: 0.624000; val_acc: 0.324000
(Epoch 18 / 25) train acc: 0.610000; val_acc: 0.325000
(Epoch 19 / 25) train acc: 0.620000; val_acc: 0.327000
(Epoch 20 / 25) train acc: 0.662000; val_acc: 0.340000
(Iteration 101 / 125) loss: 1.296187
(Epoch 21 / 25) train acc: 0.688000; val_acc: 0.323000
(Epoch 22 / 25) train acc: 0.708000; val_acc: 0.323000
(Epoch 23 / 25) train acc: 0.742000; val_acc: 0.345000
(Epoch 24 / 25) train acc: 0.756000; val_acc: 0.325000
(Epoch 25 / 25) train acc: 0.782000; val_acc: 0.348000
```

In [7]:

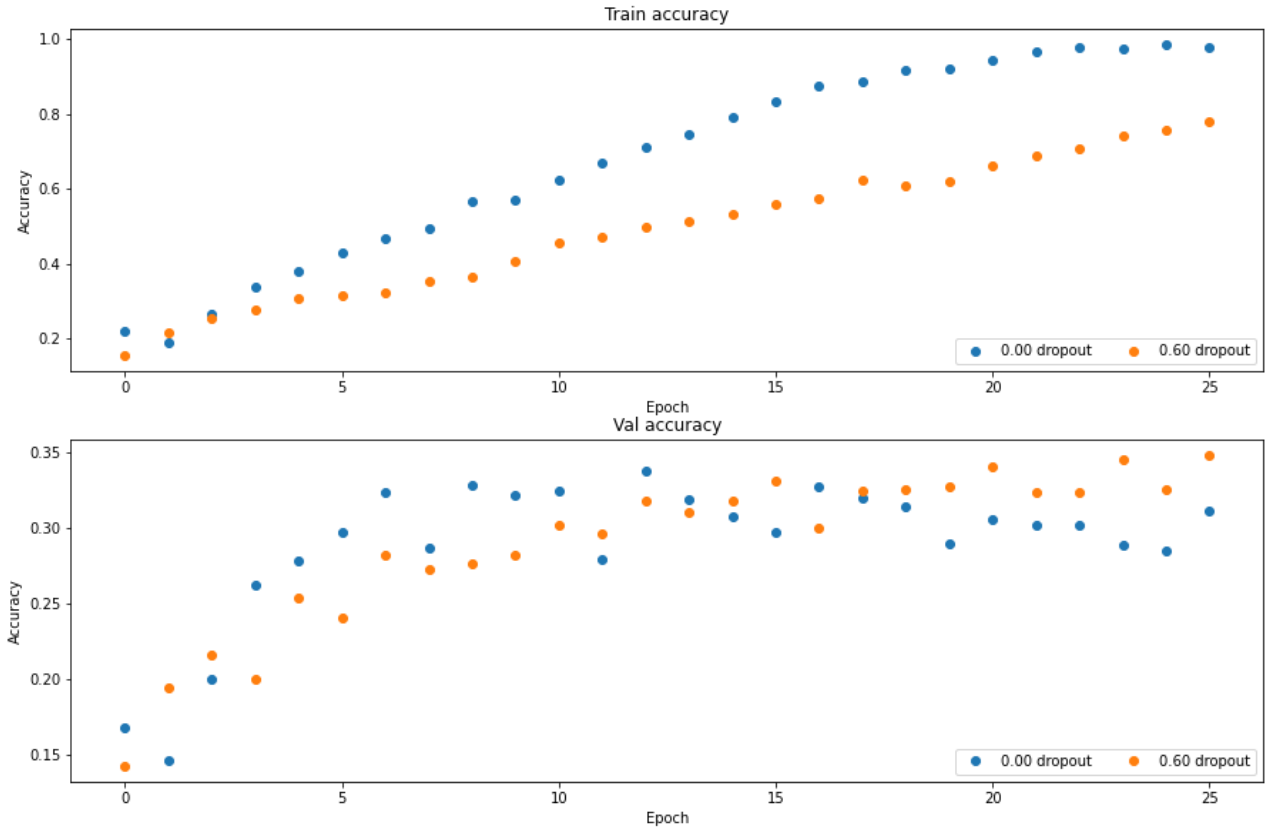
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropo
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



Question

Based off the results of this experiment, is dropout performing regularization? Explain your answer.

Answer:

Yes. From the accuracy figure we can find out the overfitting is reduced during the training.

Final part of the assignment

Get over 55% validation accuracy on CIFAR-10 by using the layers you have implemented. You will be graded according to the following equation:

min(floor((X - 32%)) / 28%, 1) where if you get 60% or higher validation accuracy, you get full points.

In [14]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #

optimizer = 'rmsprop'

layer_dims = [600, 600, 600, 600]
dropouts = [0.8]#[0.5, 0.6, 0.7, 0.8, 0.9]
```

```
weight_scales = [1e-2]#[1e-1, 5e-2, 1e-2, 5e-3, 1e-3]
learning_rates = [1e-3]#[1e-2, 5e-3, 1e-3, 5e-4, 1e-4]
regs = [0.3]#[0.0, 0.1, 0.2, 0.3]
lr_decay = 0.95

solvers = {}
for dropout in dropouts:
    for lr in learning_rates:
        for reg in regs:
            for w in weight_scales:
                index = (dropout, lr, reg, w)
                model = FullyConnectedNet(layer_dims,
                          weight_scale = w,
                          dropout = dropout,
                          reg = reg,
                          use_batchnorm=True)

                solver = Solver(model, data,
                            num_epochs=80, batch_size=100,
                            update_rule=optimizer,
                            optim_config={
                          'learning_rate': lr,
                            },
                            lr_decay=lr_decay,
                            verbose=True, print_every=1000)

                solver.train()
                solvers[index] = solver
                best_val = np.amax(solvers[index].val_acc_history)
                best_train = np.amax(solvers[index].train_acc_history)
                print('dropout:', dropout,'lr:',lr,'reg:',reg,'weight:',w,'best_


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

```
(Iteration 1 / 39200) loss: 46.232648
(Epoch 0 / 80) train acc: 0.110000; val_acc: 0.103000
(Epoch 1 / 80) train acc: 0.113000; val_acc: 0.108000
(Epoch 2 / 80) train acc: 0.110000; val_acc: 0.093000
(Iteration 1001 / 39200) loss: 3.014885
(Epoch 3 / 80) train acc: 0.115000; val_acc: 0.147000
(Epoch 4 / 80) train acc: 0.162000; val_acc: 0.142000
(Iteration 2001 / 39200) loss: 2.723789
(Epoch 5 / 80) train acc: 0.134000; val_acc: 0.162000
(Epoch 6 / 80) train acc: 0.153000; val_acc: 0.191000
(Iteration 3001 / 39200) loss: 2.652701
(Epoch 7 / 80) train acc: 0.148000; val_acc: 0.152000
(Epoch 8 / 80) train acc: 0.098000; val_acc: 0.127000
(Iteration 4001 / 39200) loss: 2.612825
(Epoch 9 / 80) train acc: 0.177000; val_acc: 0.201000
(Epoch 10 / 80) train acc: 0.197000; val_acc: 0.213000
(Iteration 5001 / 39200) loss: 2.430625
(Epoch 11 / 80) train acc: 0.212000; val_acc: 0.230000
(Epoch 12 / 80) train acc: 0.226000; val_acc: 0.220000
(Iteration 6001 / 39200) loss: 2.290217
(Epoch 13 / 80) train acc: 0.241000; val_acc: 0.241000
(Epoch 14 / 80) train acc: 0.239000; val_acc: 0.226000
(Iteration 7001 / 39200) loss: 2.184822
(Epoch 15 / 80) train acc: 0.201000; val_acc: 0.206000
(Epoch 16 / 80) train acc: 0.210000; val_acc: 0.211000
```

```
(Iteration 8001 / 39200) loss: 2.272797
(Epoch 17 / 80) train acc: 0.224000; val_acc: 0.254000
(Epoch 18 / 80) train acc: 0.228000; val_acc: 0.268000
(Iteration 9001 / 39200) loss: 2.373723
(Epoch 19 / 80) train acc: 0.292000; val_acc: 0.248000
(Epoch 20 / 80) train acc: 0.260000; val_acc: 0.288000
(Iteration 10001 / 39200) loss: 2.112414
(Epoch 21 / 80) train acc: 0.282000; val_acc: 0.312000
(Epoch 22 / 80) train acc: 0.312000; val_acc: 0.305000
(Iteration 11001 / 39200) loss: 1.991591
(Epoch 23 / 80) train acc: 0.356000; val_acc: 0.354000
(Epoch 24 / 80) train acc: 0.351000; val_acc: 0.364000
(Iteration 12001 / 39200) loss: 2.046677
(Epoch 25 / 80) train acc: 0.380000; val_acc: 0.392000
(Epoch 26 / 80) train acc: 0.358000; val_acc: 0.375000
(Iteration 13001 / 39200) loss: 1.991193
(Epoch 27 / 80) train acc: 0.372000; val_acc: 0.373000
(Epoch 28 / 80) train acc: 0.355000; val_acc: 0.366000
(Iteration 14001 / 39200) loss: 2.086809
(Epoch 29 / 80) train acc: 0.386000; val_acc: 0.382000
(Epoch 30 / 80) train acc: 0.377000; val_acc: 0.382000
(Iteration 15001 / 39200) loss: 1.972225
(Epoch 31 / 80) train acc: 0.383000; val_acc: 0.385000
(Epoch 32 / 80) train acc: 0.377000; val_acc: 0.398000
(Iteration 16001 / 39200) loss: 2.031859
(Epoch 33 / 80) train acc: 0.397000; val_acc: 0.415000
(Epoch 34 / 80) train acc: 0.380000; val_acc: 0.397000
(Iteration 17001 / 39200) loss: 1.904531
(Epoch 35 / 80) train acc: 0.403000; val_acc: 0.429000
(Epoch 36 / 80) train acc: 0.418000; val_acc: 0.402000
(Iteration 18001 / 39200) loss: 1.916423
(Epoch 37 / 80) train acc: 0.404000; val_acc: 0.430000
(Epoch 38 / 80) train acc: 0.426000; val_acc: 0.424000
(Iteration 19001 / 39200) loss: 2.013001
(Epoch 39 / 80) train acc: 0.428000; val_acc: 0.434000
(Epoch 40 / 80) train acc: 0.417000; val_acc: 0.442000
(Iteration 20001 / 39200) loss: 1.817271
(Epoch 41 / 80) train acc: 0.452000; val_acc: 0.457000
(Epoch 42 / 80) train acc: 0.437000; val_acc: 0.462000
(Iteration 21001 / 39200) loss: 1.929943
(Epoch 43 / 80) train acc: 0.477000; val_acc: 0.453000
(Epoch 44 / 80) train acc: 0.409000; val_acc: 0.435000
(Iteration 22001 / 39200) loss: 1.891632
(Epoch 45 / 80) train acc: 0.449000; val_acc: 0.457000
(Epoch 46 / 80) train acc: 0.451000; val_acc: 0.449000
(Iteration 23001 / 39200) loss: 1.834037
(Epoch 47 / 80) train acc: 0.435000; val_acc: 0.456000
(Epoch 48 / 80) train acc: 0.465000; val_acc: 0.483000
(Iteration 24001 / 39200) loss: 1.838202
(Epoch 49 / 80) train acc: 0.472000; val_acc: 0.442000
(Epoch 50 / 80) train acc: 0.486000; val_acc: 0.473000
(Epoch 51 / 80) train acc: 0.479000; val_acc: 0.483000
(Iteration 25001 / 39200) loss: 1.853627
(Epoch 52 / 80) train acc: 0.447000; val_acc: 0.462000
(Epoch 53 / 80) train acc: 0.501000; val_acc: 0.497000
(Iteration 26001 / 39200) loss: 1.755954
(Epoch 54 / 80) train acc: 0.474000; val_acc: 0.481000
(Epoch 55 / 80) train acc: 0.499000; val_acc: 0.495000
(Iteration 27001 / 39200) loss: 1.511890
(Epoch 56 / 80) train acc: 0.498000; val_acc: 0.499000
(Epoch 57 / 80) train acc: 0.499000; val_acc: 0.483000
(Iteration 28001 / 39200) loss: 1.661805
(Epoch 58 / 80) train acc: 0.479000; val_acc: 0.488000
(Epoch 59 / 80) train acc: 0.504000; val_acc: 0.484000
(Iteration 29001 / 39200) loss: 1.513723
```

```
(Epoch 60 / 80) train acc: 0.508000; val_acc: 0.514000
(Epoch 61 / 80) train acc: 0.531000; val_acc: 0.517000
(Iteration 30001 / 39200) loss: 1.736998
(Epoch 62 / 80) train acc: 0.538000; val_acc: 0.497000
(Epoch 63 / 80) train acc: 0.512000; val_acc: 0.503000
(Iteration 31001 / 39200) loss: 1.556728
(Epoch 64 / 80) train acc: 0.548000; val_acc: 0.526000
(Epoch 65 / 80) train acc: 0.537000; val_acc: 0.500000
(Iteration 32001 / 39200) loss: 1.635866
(Epoch 66 / 80) train acc: 0.569000; val_acc: 0.505000
(Epoch 67 / 80) train acc: 0.520000; val_acc: 0.517000
(Iteration 33001 / 39200) loss: 1.638790
(Epoch 68 / 80) train acc: 0.538000; val_acc: 0.514000
(Epoch 69 / 80) train acc: 0.531000; val_acc: 0.499000
(Iteration 34001 / 39200) loss: 1.528642
(Epoch 70 / 80) train acc: 0.586000; val_acc: 0.510000
(Epoch 71 / 80) train acc: 0.555000; val_acc: 0.515000
(Iteration 35001 / 39200) loss: 1.628689
(Epoch 72 / 80) train acc: 0.564000; val_acc: 0.529000
(Epoch 73 / 80) train acc: 0.580000; val_acc: 0.514000
(Iteration 36001 / 39200) loss: 1.479091
(Epoch 74 / 80) train acc: 0.567000; val_acc: 0.522000
(Epoch 75 / 80) train acc: 0.582000; val_acc: 0.518000
(Iteration 37001 / 39200) loss: 1.654059
(Epoch 76 / 80) train acc: 0.592000; val_acc: 0.510000
(Epoch 77 / 80) train acc: 0.588000; val_acc: 0.516000
(Iteration 38001 / 39200) loss: 1.500045
(Epoch 78 / 80) train acc: 0.563000; val_acc: 0.525000
(Epoch 79 / 80) train acc: 0.592000; val_acc: 0.542000
(Iteration 39001 / 39200) loss: 1.420499
(Epoch 80 / 80) train acc: 0.599000; val_acc: 0.526000
dropout: 0.8 lr: 0.001 reg: 0.3 weight: 0.01 best_train: 0.599 best_val: 0.542
```

In [ ]: