Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive over 60% 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 [17]:

```
## 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_numerica
l_gradient_array
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-i
n-ipython
%load_ext autoreload
%autoreload 2

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

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

In [18]:
```
# 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,)
```

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 [20]:
```
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).me
an())
```

```
('Running tests with p = ', 0.3)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 23.311412482547)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.30092)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.6)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 6.652111061656034)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.600916)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.75)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 3.3289192472660556)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.75024)
('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 [21]: 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, dr
         opout_param)[0], x, dout)

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

```
('dx relative error: ', 5.445607071257824e-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 [24]:
```
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=Fa
lse, h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads
[name])))
  print('\n')
```

```
('Running check with dropout = ', 0)
('Initial loss: ', 2.298949958544263)
W1 relative error: 1.87119719924e-06
W2 relative error: 3.75955387515e-07
W3 relative error: 4.41558101425e-08
b1 relative error: 2.73991000184e-08
b2 relative error: 3.46941444851e-09
b3 relative error: 1.38519530797e-10


('Running check with dropout = ', 0.25)
('Initial loss: ', 2.2597611820017107)
W1 relative error: 5.90756092267e-09
W2 relative error: 1.30607120549e-07
W3 relative error: 1.2081297763e-08
b1 relative error: 5.47914274408e-10
b2 relative error: 1.23932214532e-07
b3 relative error: 5.96589419451e-11


('Running check with dropout = ', 0.5)
('Initial loss: ', 2.298135001122367)
W1 relative error: 2.19459352143e-08
W2 relative error: 1.8332201706e-08
W3 relative error: 2.0893611084e-08
b1 relative error: 6.06139640116e-10
b2 relative error: 1.63235192309e-09
b3 relative error: 1.2861046903e-10
```

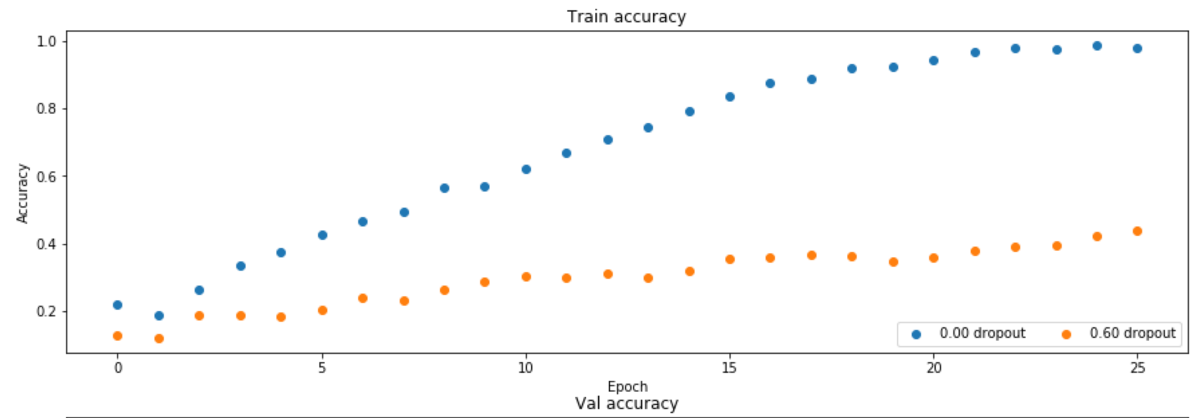
Dropout as a regularizer

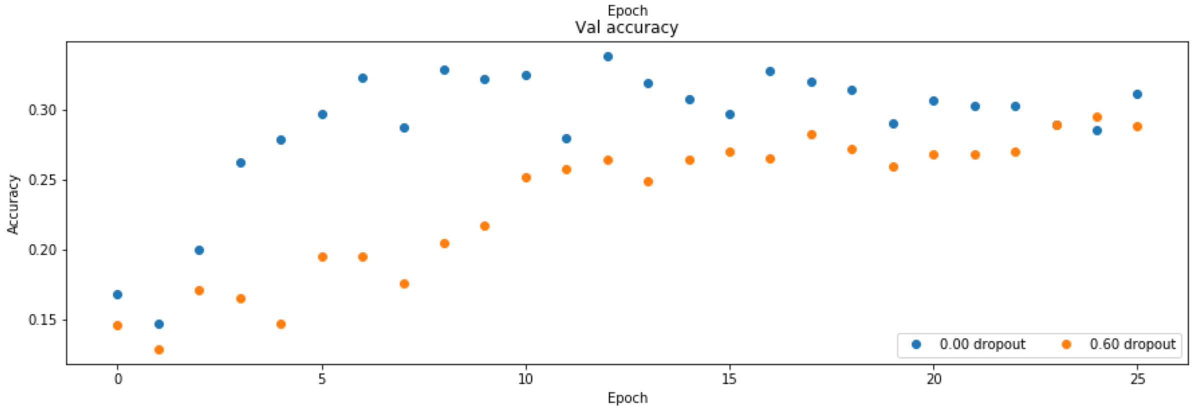
In class, we claimed that dropout acts as a regularizer by effectively bagging. To check this, we will train two small networks, one with dropout and one without dropout.

In [25]:
```
# 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: 2.301219
(Epoch 0 / 25) train acc: 0.132000; val_acc: 0.146000
(Epoch 1 / 25) train acc: 0.122000; val_acc: 0.129000
(Epoch 2 / 25) train acc: 0.190000; val_acc: 0.171000
(Epoch 3 / 25) train acc: 0.190000; val_acc: 0.165000
(Epoch 4 / 25) train acc: 0.186000; val_acc: 0.147000
(Epoch 5 / 25) train acc: 0.206000; val_acc: 0.195000
(Epoch 6 / 25) train acc: 0.240000; val_acc: 0.195000
(Epoch 7 / 25) train acc: 0.234000; val_acc: 0.176000
(Epoch 8 / 25) train acc: 0.264000; val_acc: 0.205000
(Epoch 9 / 25) train acc: 0.288000; val_acc: 0.217000
(Epoch 10 / 25) train acc: 0.306000; val_acc: 0.252000
(Epoch 11 / 25) train acc: 0.302000; val_acc: 0.257000
(Epoch 12 / 25) train acc: 0.312000; val_acc: 0.264000
(Epoch 13 / 25) train acc: 0.300000; val_acc: 0.249000
(Epoch 14 / 25) train acc: 0.322000; val_acc: 0.264000
(Epoch 15 / 25) train acc: 0.358000; val_acc: 0.270000
(Epoch 16 / 25) train acc: 0.360000; val_acc: 0.265000
(Epoch 17 / 25) train acc: 0.368000; val_acc: 0.282000
(Epoch 18 / 25) train acc: 0.364000; val_acc: 0.272000
(Epoch 19 / 25) train acc: 0.350000; val_acc: 0.259000
(Epoch 20 / 25) train acc: 0.360000; val_acc: 0.268000
(Iteration 101 / 125) loss: 1.839906
(Epoch 21 / 25) train acc: 0.380000; val_acc: 0.268000
(Epoch 22 / 25) train acc: 0.392000; val_acc: 0.270000
(Epoch 23 / 25) train acc: 0.396000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.424000; val_acc: 0.295000
(Epoch 25 / 25) train acc: 0.440000; val_acc: 0.288000
```

In [26]:
```
# 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'
 % dropout)
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:

Without dropout, we can see that trianing accuracy ~97% but validation accuracy is 31%. This shows that the model is overfitting. After dropout, training accuracy and Validation accurac ~ 40% and 30%. The training accuracy is comparable to validation accuracy. Therefore, the model is not overfitting which proves that dropout actually is sort of a regularization.

Final part of the assignment

Get over 60% 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 [56]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=50, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 24500) loss: 10.367319
(Epoch 0 / 50) train acc: 0.112000; val_acc: 0.132000
(Epoch 1 / 50) train acc: 0.453000; val_acc: 0.412000
(Iteration 501 / 24500) loss: 3.678347
(Epoch 2 / 50) train acc: 0.449000; val_acc: 0.453000
(Iteration 1001 / 24500) loss: 2.483102
(Epoch 3 / 50) train acc: 0.476000; val_acc: 0.479000
(Iteration 1501 / 24500) loss: 2.405102
(Epoch 4 / 50) train acc: 0.500000; val_acc: 0.472000
(Iteration 2001 / 24500) loss: 1.957018
(Epoch 5 / 50) train acc: 0.475000; val_acc: 0.484000
(Iteration 2501 / 24500) loss: 1.784672
(Epoch 6 / 50) train acc: 0.500000; val_acc: 0.480000
(Iteration 3001 / 24500) loss: 1.740386
(Epoch 7 / 50) train acc: 0.517000; val_acc: 0.502000
(Iteration 3501 / 24500) loss: 1.752235
(Epoch 8 / 50) train acc: 0.545000; val_acc: 0.509000
(Iteration 4001 / 24500) loss: 1.818646
(Epoch 9 / 50) train acc: 0.527000; val_acc: 0.504000
(Iteration 4501 / 24500) loss: 1.871535
(Epoch 10 / 50) train acc: 0.562000; val_acc: 0.524000
(Iteration 5001 / 24500) loss: 1.663305
(Epoch 11 / 50) train acc: 0.567000; val_acc: 0.533000
(Iteration 5501 / 24500) loss: 1.637834
(Epoch 12 / 50) train acc: 0.591000; val_acc: 0.533000
(Iteration 6001 / 24500) loss: 1.482642
(Epoch 13 / 50) train acc: 0.621000; val_acc: 0.562000
(Iteration 6501 / 24500) loss: 1.545891
(Epoch 14 / 50) train acc: 0.584000; val_acc: 0.575000
(Iteration 7001 / 24500) loss: 1.558609
(Epoch 15 / 50) train acc: 0.650000; val_acc: 0.561000
(Iteration 7501 / 24500) loss: 1.457756
(Epoch 16 / 50) train acc: 0.651000; val_acc: 0.572000
(Iteration 8001 / 24500) loss: 1.329249
(Epoch 17 / 50) train acc: 0.642000; val_acc: 0.577000
(Iteration 8501 / 24500) loss: 1.236984
(Epoch 18 / 50) train acc: 0.682000; val_acc: 0.564000
(Iteration 9001 / 24500) loss: 1.492818
(Epoch 19 / 50) train acc: 0.683000; val_acc: 0.579000
(Iteration 9501 / 24500) loss: 1.138019
(Epoch 20 / 50) train acc: 0.700000; val_acc: 0.570000
(Iteration 10001 / 24500) loss: 1.370272
(Epoch 21 / 50) train acc: 0.681000; val_acc: 0.567000
(Iteration 10501 / 24500) loss: 1.246665
(Epoch 22 / 50) train acc: 0.688000; val_acc: 0.580000
(Iteration 11001 / 24500) loss: 1.193318
(Epoch 23 / 50) train acc: 0.693000; val_acc: 0.580000
(Iteration 11501 / 24500) loss: 1.133695
(Epoch 24 / 50) train acc: 0.703000; val_acc: 0.587000
(Iteration 12001 / 24500) loss: 1.189112
(Epoch 25 / 50) train acc: 0.707000; val_acc: 0.584000
(Iteration 12501 / 24500) loss: 1.184437
(Epoch 26 / 50) train acc: 0.731000; val_acc: 0.585000
(Iteration 13001 / 24500) loss: 1.185885
(Epoch 27 / 50) train acc: 0.754000; val_acc: 0.582000
(Iteration 13501 / 24500) loss: 1.130842
(Epoch 28 / 50) train acc: 0.768000; val_acc: 0.586000
```

```
(Iteration 14001 / 24500) loss: 0.868604
(Epoch 29 / 50) train acc: 0.773000; val_acc: 0.592000
(Iteration 14501 / 24500) loss: 0.922187
(Epoch 30 / 50) train acc: 0.744000; val_acc: 0.584000
(Iteration 15001 / 24500) loss: 1.206035
(Epoch 31 / 50) train acc: 0.778000; val_acc: 0.592000
(Iteration 15501 / 24500) loss: 1.383444
(Epoch 32 / 50) train acc: 0.743000; val_acc: 0.592000
(Iteration 16001 / 24500) loss: 1.157079
(Epoch 33 / 50) train acc: 0.796000; val_acc: 0.587000
(Iteration 16501 / 24500) loss: 1.108257
(Epoch 34 / 50) train acc: 0.777000; val_acc: 0.602000
(Iteration 17001 / 24500) loss: 0.997282
(Epoch 35 / 50) train acc: 0.793000; val_acc: 0.597000
(Iteration 17501 / 24500) loss: 0.914432
(Epoch 36 / 50) train acc: 0.798000; val_acc: 0.596000
(Iteration 18001 / 24500) loss: 0.956723
(Epoch 37 / 50) train acc: 0.811000; val_acc: 0.583000
(Iteration 18501 / 24500) loss: 1.203487
(Epoch 38 / 50) train acc: 0.792000; val_acc: 0.599000
(Iteration 19001 / 24500) loss: 1.036741
(Epoch 39 / 50) train acc: 0.793000; val_acc: 0.600000
(Iteration 19501 / 24500) loss: 0.952084
(Epoch 40 / 50) train acc: 0.828000; val_acc: 0.595000
(Iteration 20001 / 24500) loss: 0.871738
(Epoch 41 / 50) train acc: 0.818000; val_acc: 0.595000
(Iteration 20501 / 24500) loss: 0.961417
(Epoch 42 / 50) train acc: 0.821000; val_acc: 0.593000
(Iteration 21001 / 24500) loss: 0.945501
(Epoch 43 / 50) train acc: 0.816000; val_acc: 0.594000
(Iteration 21501 / 24500) loss: 0.784905
(Epoch 44 / 50) train acc: 0.845000; val_acc: 0.593000
(Iteration 22001 / 24500) loss: 0.953447
(Epoch 45 / 50) train acc: 0.823000; val_acc: 0.598000
(Iteration 22501 / 24500) loss: 0.893920
(Epoch 46 / 50) train acc: 0.823000; val_acc: 0.602000
(Iteration 23001 / 24500) loss: 0.797951
(Epoch 47 / 50) train acc: 0.835000; val_acc: 0.598000
(Iteration 23501 / 24500) loss: 0.949376
(Epoch 48 / 50) train acc: 0.797000; val_acc: 0.596000
(Iteration 24001 / 24500) loss: 0.955032
(Epoch 49 / 50) train acc: 0.836000; val_acc: 0.593000
(Epoch 50 / 50) train acc: 0.833000; val_acc: 0.594000
```

In [62]:
```
# ============================================================== #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ============================================================== #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=34, batch_size=128,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ============================================================== #
# END YOUR CODE HERE
# ============================================================== #
```

```
(Iteration 1 / 12988) loss: 10.345334
(Epoch 0 / 34) train acc: 0.116000; val_acc: 0.107000
(Epoch 1 / 34) train acc: 0.481000; val_acc: 0.433000
(Iteration 501 / 12988) loss: 3.388941
(Epoch 2 / 34) train acc: 0.470000; val_acc: 0.475000
(Iteration 1001 / 12988) loss: 2.470986
(Epoch 3 / 34) train acc: 0.492000; val_acc: 0.495000
(Iteration 1501 / 12988) loss: 1.965256
(Epoch 4 / 34) train acc: 0.494000; val_acc: 0.500000
(Epoch 5 / 34) train acc: 0.495000; val_acc: 0.480000
(Iteration 2001 / 12988) loss: 1.982240
(Epoch 6 / 34) train acc: 0.540000; val_acc: 0.498000
(Iteration 2501 / 12988) loss: 1.784661
(Epoch 7 / 34) train acc: 0.538000; val_acc: 0.509000
(Iteration 3001 / 12988) loss: 1.667780
(Epoch 8 / 34) train acc: 0.559000; val_acc: 0.510000
(Epoch 9 / 34) train acc: 0.552000; val_acc: 0.537000
(Iteration 3501 / 12988) loss: 1.809124
(Epoch 10 / 34) train acc: 0.569000; val_acc: 0.546000
(Iteration 4001 / 12988) loss: 1.487143
(Epoch 11 / 34) train acc: 0.591000; val_acc: 0.531000
(Iteration 4501 / 12988) loss: 1.475054
(Epoch 12 / 34) train acc: 0.569000; val_acc: 0.547000
(Epoch 13 / 34) train acc: 0.622000; val_acc: 0.556000
(Iteration 5001 / 12988) loss: 1.589821
(Epoch 14 / 34) train acc: 0.621000; val_acc: 0.559000
(Iteration 5501 / 12988) loss: 1.355803
(Epoch 15 / 34) train acc: 0.611000; val_acc: 0.562000
(Iteration 6001 / 12988) loss: 1.367220
(Epoch 16 / 34) train acc: 0.625000; val_acc: 0.569000
(Epoch 17 / 34) train acc: 0.677000; val_acc: 0.574000
(Iteration 6501 / 12988) loss: 1.300967
(Epoch 18 / 34) train acc: 0.659000; val_acc: 0.588000
(Iteration 7001 / 12988) loss: 1.230683
(Epoch 19 / 34) train acc: 0.684000; val_acc: 0.574000
(Iteration 7501 / 12988) loss: 1.341910
(Epoch 20 / 34) train acc: 0.701000; val_acc: 0.574000
(Iteration 8001 / 12988) loss: 1.442092
(Epoch 21 / 34) train acc: 0.710000; val_acc: 0.575000
(Epoch 22 / 34) train acc: 0.728000; val_acc: 0.585000
(Iteration 8501 / 12988) loss: 1.237475
(Epoch 23 / 34) train acc: 0.727000; val_acc: 0.592000
(Iteration 9001 / 12988) loss: 1.103268
(Epoch 24 / 34) train acc: 0.698000; val_acc: 0.585000
(Iteration 9501 / 12988) loss: 1.191798
(Epoch 25 / 34) train acc: 0.767000; val_acc: 0.598000
(Epoch 26 / 34) train acc: 0.736000; val_acc: 0.595000
(Iteration 10001 / 12988) loss: 1.130865
(Epoch 27 / 34) train acc: 0.750000; val_acc: 0.594000
(Iteration 10501 / 12988) loss: 1.070354
(Epoch 28 / 34) train acc: 0.747000; val_acc: 0.594000
(Iteration 11001 / 12988) loss: 0.901335
(Epoch 29 / 34) train acc: 0.779000; val_acc: 0.593000
(Epoch 30 / 34) train acc: 0.762000; val_acc: 0.589000
(Iteration 11501 / 12988) loss: 1.099542
(Epoch 31 / 34) train acc: 0.784000; val_acc: 0.607000
(Iteration 12001 / 12988) loss: 0.901268
```

```
(Epoch 32 / 34) train acc: 0.769000; val_acc: 0.603000
(Iteration 12501 / 12988) loss: 0.901700
(Epoch 33 / 34) train acc: 0.767000; val_acc: 0.582000
(Epoch 34 / 34) train acc: 0.815000; val_acc: 0.599000
```

In [64]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=40, batch_size=128,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 15280) loss: 10.338546
(Epoch 0 / 40) train acc: 0.097000; val_acc: 0.093000
(Epoch 1 / 40) train acc: 0.434000; val_acc: 0.429000
(Iteration 501 / 15280) loss: 3.395782
(Epoch 2 / 40) train acc: 0.440000; val_acc: 0.459000
(Iteration 1001 / 15280) loss: 2.147055
(Epoch 3 / 40) train acc: 0.494000; val_acc: 0.468000
(Iteration 1501 / 15280) loss: 2.026479
(Epoch 4 / 40) train acc: 0.498000; val_acc: 0.482000
(Epoch 5 / 40) train acc: 0.494000; val_acc: 0.507000
(Iteration 2001 / 15280) loss: 1.818237
(Epoch 6 / 40) train acc: 0.536000; val_acc: 0.505000
(Iteration 2501 / 15280) loss: 1.817284
(Epoch 7 / 40) train acc: 0.538000; val_acc: 0.528000
(Iteration 3001 / 15280) loss: 1.670779
(Epoch 8 / 40) train acc: 0.550000; val_acc: 0.534000
(Epoch 9 / 40) train acc: 0.582000; val_acc: 0.538000
(Iteration 3501 / 15280) loss: 1.703494
(Epoch 10 / 40) train acc: 0.538000; val_acc: 0.535000
(Iteration 4001 / 15280) loss: 1.724836
(Epoch 11 / 40) train acc: 0.587000; val_acc: 0.547000
(Iteration 4501 / 15280) loss: 1.524487
(Epoch 12 / 40) train acc: 0.567000; val_acc: 0.548000
(Epoch 13 / 40) train acc: 0.607000; val_acc: 0.558000
(Iteration 5001 / 15280) loss: 1.630233
(Epoch 14 / 40) train acc: 0.611000; val_acc: 0.545000
(Iteration 5501 / 15280) loss: 1.350028
(Epoch 15 / 40) train acc: 0.643000; val_acc: 0.553000
(Iteration 6001 / 15280) loss: 1.298213
(Epoch 16 / 40) train acc: 0.678000; val_acc: 0.552000
(Epoch 17 / 40) train acc: 0.671000; val_acc: 0.583000
(Iteration 6501 / 15280) loss: 1.299723
(Epoch 18 / 40) train acc: 0.672000; val_acc: 0.575000
(Iteration 7001 / 15280) loss: 1.384886
(Epoch 19 / 40) train acc: 0.646000; val_acc: 0.585000
(Iteration 7501 / 15280) loss: 1.313679
(Epoch 20 / 40) train acc: 0.677000; val_acc: 0.577000
(Iteration 8001 / 15280) loss: 1.364944
(Epoch 21 / 40) train acc: 0.688000; val_acc: 0.580000
(Epoch 22 / 40) train acc: 0.732000; val_acc: 0.586000
(Iteration 8501 / 15280) loss: 1.311282
(Epoch 23 / 40) train acc: 0.725000; val_acc: 0.589000
(Iteration 9001 / 15280) loss: 1.216050
(Epoch 24 / 40) train acc: 0.724000; val_acc: 0.577000
(Iteration 9501 / 15280) loss: 1.147909
(Epoch 25 / 40) train acc: 0.756000; val_acc: 0.598000
(Epoch 26 / 40) train acc: 0.718000; val_acc: 0.608000
(Iteration 10001 / 15280) loss: 1.139750
(Epoch 27 / 40) train acc: 0.744000; val_acc: 0.583000
(Iteration 10501 / 15280) loss: 0.973199
(Epoch 28 / 40) train acc: 0.735000; val_acc: 0.590000
(Iteration 11001 / 15280) loss: 1.104056
(Epoch 29 / 40) train acc: 0.763000; val_acc: 0.591000
(Epoch 30 / 40) train acc: 0.791000; val_acc: 0.582000
(Iteration 11501 / 15280) loss: 0.944136
(Epoch 31 / 40) train acc: 0.771000; val_acc: 0.588000
(Iteration 12001 / 15280) loss: 1.012222
```

```
(Epoch 32 / 40) train acc: 0.798000; val_acc: 0.585000
(Iteration 12501 / 15280) loss: 1.036070
(Epoch 33 / 40) train acc: 0.787000; val_acc: 0.591000
(Epoch 34 / 40) train acc: 0.788000; val_acc: 0.585000
(Iteration 13001 / 15280) loss: 0.927312
(Epoch 35 / 40) train acc: 0.784000; val_acc: 0.598000
(Iteration 13501 / 15280) loss: 1.047863
(Epoch 36 / 40) train acc: 0.794000; val_acc: 0.594000
(Iteration 14001 / 15280) loss: 1.043726
(Epoch 37 / 40) train acc: 0.800000; val_acc: 0.590000
(Iteration 14501 / 15280) loss: 0.920529
(Epoch 38 / 40) train acc: 0.810000; val_acc: 0.588000
(Epoch 39 / 40) train acc: 0.826000; val_acc: 0.588000
(Iteration 15001 / 15280) loss: 0.863095
(Epoch 40 / 40) train acc: 0.824000; val_acc: 0.594000
```

In [ ]:
```
Validation Accuracy of ~60% achived in earlie Epochs.
```