ECE C147/247 HW4 Q3: 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.

`utils` 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`.

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 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))))
```

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:  10.001329958949242
Mean of train-time output:  10.000677630103738
Mean of test-time output:  10.001329958949242
Fraction of train-time output set to zero:  0.300204
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.001329958949242
Mean of train-time output:  9.995013082981687
Mean of test-time output:  10.001329958949242
Fraction of train-time output set to zero:  0.600344
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.001329958949242
Mean of train-time output:  9.99724972751136
Mean of test-time output:  10.001329958949242
Fraction of train-time output set to zero:  0.750256
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_param)[0],

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

```
dx relative error:  1.8929063206183813e-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=1e-5)
    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: 2.5272575344376073e-07
W2 relative error: 1.8875602996866792e-05
W3 relative error: 2.3446001188064074e-07
b1 relative error: 1.3413524910372306e-07
b2 relative error: 5.051339805546953e-08
b3 relative error: 1.4926760614954125e-10


Running check with dropout =  0.25
Initial loss:  2.29898614757146
W1 relative error: 9.737733598900162e-07
W2 relative error: 5.073657932383196e-08
W3 relative error: 2.8870225716091813e-08
b1 relative error: 9.618747087456625e-09
b2 relative error: 1.897778283870511e-09
b3 relative error: 8.933554101600298e-11


Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.553387957138422e-08
W2 relative error: 1.8475263967389784e-08
W3 relative error: 4.3413247403122424e-07
b1 relative error: 1.872462967441693e-08
b2 relative error: 5.045591219274328e-09
b3 relative error: 1.2715825087959627e-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 [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: 2.306395
(Epoch 0 / 25) train acc: 0.120000; val_acc: 0.131000
(Epoch 1 / 25) train acc: 0.170000; val_acc: 0.166000
(Epoch 2 / 25) train acc: 0.246000; val_acc: 0.208000
(Epoch 3 / 25) train acc: 0.240000; val_acc: 0.193000
(Epoch 4 / 25) train acc: 0.234000; val_acc: 0.203000
(Epoch 5 / 25) train acc: 0.234000; val_acc: 0.207000
(Epoch 6 / 25) train acc: 0.238000; val_acc: 0.202000
(Epoch 7 / 25) train acc: 0.276000; val_acc: 0.224000
(Epoch 8 / 25) train acc: 0.288000; val_acc: 0.249000
(Epoch 9 / 25) train acc: 0.314000; val_acc: 0.250000
(Epoch 10 / 25) train acc: 0.324000; val_acc: 0.267000
(Epoch 11 / 25) train acc: 0.360000; val_acc: 0.263000
(Epoch 12 / 25) train acc: 0.360000; val_acc: 0.293000
(Epoch 13 / 25) train acc: 0.352000; val_acc: 0.266000
(Epoch 14 / 25) train acc: 0.366000; val_acc: 0.273000
(Epoch 15 / 25) train acc: 0.390000; val_acc: 0.280000
(Epoch 16 / 25) train acc: 0.438000; val_acc: 0.294000
(Epoch 17 / 25) train acc: 0.442000; val_acc: 0.293000
(Epoch 18 / 25) train acc: 0.416000; val_acc: 0.303000
(Epoch 19 / 25) train acc: 0.400000; val_acc: 0.271000
(Epoch 20 / 25) train acc: 0.384000; val_acc: 0.280000
(Iteration 101 / 125) loss: 1.881304
(Epoch 21 / 25) train acc: 0.412000; val_acc: 0.290000
(Epoch 22 / 25) train acc: 0.460000; val_acc: 0.303000
(Epoch 23 / 25) train acc: 0.494000; val_acc: 0.309000
(Epoch 24 / 25) train acc: 0.474000; val_acc: 0.312000
(Epoch 25 / 25) train acc: 0.488000; val_acc: 0.311000
```

In [7]:

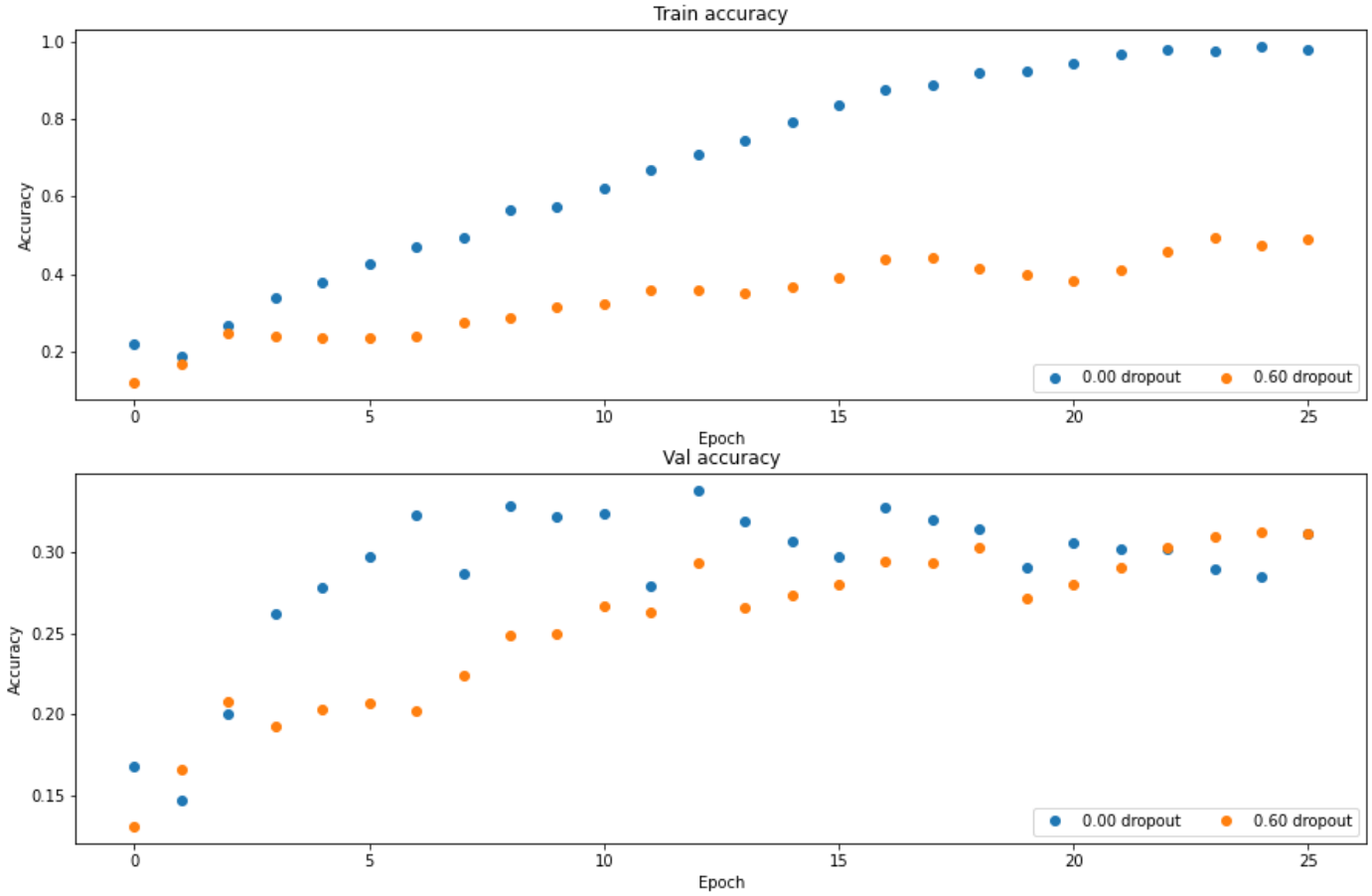
```
# Plot train and validation accuracies of the two models

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' % 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:

- Yes, it is performing regularization. From the second plot, we can find that the model with or without dropout have similar validation accuracies. While in the first plot, the model without dropout has significantly higher training accuracies than the model with dropout. It means that the additional training accuracies of the model without dropout is overfitting, and the dropout regularized it.

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 [8]:

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

hidden_dims = [600, 600, 600, 600]
weight_scale = 0.04
dropout = 0.2
learning_rate = 3e-3
lr_decay = 0.95
update_rule = 'adam'

model = FullyConnectedNet(hidden_dims = hidden_dims, weight_scale = weight_scale, dropou
                          use_batchnorm = True, reg = 0.0)

solver = Solver(model, data,
                num_epochs = 100, batch_size = 500,
                update_rule = update_rule,
                optim_config = {
                  'learning_rate': learning_rate,
                },
                lr_decay = lr_decay,
                verbose=True, print_every = 100)
solver.train()

y_test_pred = np.argmax(model.loss(data['X_test']), axis = 1)
y_val_pred = np.argmax(model.loss(data['X_val']), axis = 1)
print("\nTest set accuracy: {}".format(np.mean(np.equal(y_test_pred, data['y_test']))))
print("Validation set accuracy: {}".format(np.mean(np.equal(y_val_pred, data['y_val']))))

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

```
(Iteration 1 / 9800) loss: 2.489383
(Epoch 0 / 100) train acc: 0.186000; val_acc: 0.200000
(Epoch 1 / 100) train acc: 0.436000; val_acc: 0.443000
(Iteration 101 / 9800) loss: 1.597815
(Epoch 2 / 100) train acc: 0.484000; val_acc: 0.498000
(Iteration 201 / 9800) loss: 1.366692
(Epoch 3 / 100) train acc: 0.537000; val_acc: 0.515000
(Iteration 301 / 9800) loss: 1.398289
(Epoch 4 / 100) train acc: 0.577000; val_acc: 0.566000
(Iteration 401 / 9800) loss: 1.248255
(Epoch 5 / 100) train acc: 0.599000; val_acc: 0.538000
(Iteration 501 / 9800) loss: 1.140919
(Epoch 6 / 100) train acc: 0.664000; val_acc: 0.548000
(Iteration 601 / 9800) loss: 1.117880
(Epoch 7 / 100) train acc: 0.613000; val_acc: 0.548000
(Iteration 701 / 9800) loss: 1.022194
(Epoch 8 / 100) train acc: 0.691000; val_acc: 0.580000
(Iteration 801 / 9800) loss: 1.053228
(Epoch 9 / 100) train acc: 0.702000; val_acc: 0.565000
(Iteration 901 / 9800) loss: 0.900552
(Epoch 10 / 100) train acc: 0.715000; val_acc: 0.581000
(Iteration 1001 / 9800) loss: 0.955007
(Epoch 11 / 100) train acc: 0.746000; val_acc: 0.593000
(Iteration 1101 / 9800) loss: 0.873861
(Epoch 12 / 100) train acc: 0.752000; val_acc: 0.579000
(Iteration 1201 / 9800) loss: 0.802197
(Epoch 13 / 100) train acc: 0.757000; val_acc: 0.585000
(Iteration 1301 / 9800) loss: 0.784768
(Epoch 14 / 100) train acc: 0.802000; val_acc: 0.597000
(Iteration 1401 / 9800) loss: 0.775939
(Epoch 15 / 100) train acc: 0.789000; val_acc: 0.583000
(Iteration 1501 / 9800) loss: 0.742718
(Epoch 16 / 100) train acc: 0.838000; val_acc: 0.596000
(Iteration 1601 / 9800) loss: 0.663400
(Epoch 17 / 100) train acc: 0.833000; val_acc: 0.585000
(Iteration 1701 / 9800) loss: 0.644984
(Epoch 18 / 100) train acc: 0.845000; val_acc: 0.597000
(Iteration 1801 / 9800) loss: 0.589513
(Epoch 19 / 100) train acc: 0.838000; val_acc: 0.597000
(Iteration 1901 / 9800) loss: 0.619806
(Epoch 20 / 100) train acc: 0.876000; val_acc: 0.595000
(Iteration 2001 / 9800) loss: 0.576475
(Epoch 21 / 100) train acc: 0.903000; val_acc: 0.586000
(Iteration 2101 / 9800) loss: 0.573623
(Epoch 22 / 100) train acc: 0.899000; val_acc: 0.600000
(Iteration 2201 / 9800) loss: 0.478417
(Epoch 23 / 100) train acc: 0.903000; val_acc: 0.585000
(Iteration 2301 / 9800) loss: 0.470228
(Epoch 24 / 100) train acc: 0.916000; val_acc: 0.599000
(Iteration 2401 / 9800) loss: 0.435533
(Epoch 25 / 100) train acc: 0.932000; val_acc: 0.588000
(Iteration 2501 / 9800) loss: 0.453185
(Epoch 26 / 100) train acc: 0.933000; val_acc: 0.596000
(Iteration 2601 / 9800) loss: 0.473915
(Epoch 27 / 100) train acc: 0.950000; val_acc: 0.603000
(Iteration 2701 / 9800) loss: 0.383685
(Epoch 28 / 100) train acc: 0.944000; val_acc: 0.606000
(Iteration 2801 / 9800) loss: 0.369992
(Epoch 29 / 100) train acc: 0.946000; val_acc: 0.595000
(Iteration 2901 / 9800) loss: 0.412054
(Epoch 30 / 100) train acc: 0.944000; val_acc: 0.594000
(Iteration 3001 / 9800) loss: 0.320266
(Epoch 31 / 100) train acc: 0.969000; val_acc: 0.600000
(Iteration 3101 / 9800) loss: 0.291068
(Epoch 32 / 100) train acc: 0.962000; val_acc: 0.598000
(Iteration 3201 / 9800) loss: 0.294330
```

```
(Epoch 33 / 100) train acc: 0.963000; val_acc: 0.608000
(Iteration 3301 / 9800) loss: 0.261495
(Epoch 34 / 100) train acc: 0.978000; val_acc: 0.597000
(Iteration 3401 / 9800) loss: 0.317429
(Epoch 35 / 100) train acc: 0.981000; val_acc: 0.609000
(Iteration 3501 / 9800) loss: 0.266649
(Epoch 36 / 100) train acc: 0.974000; val_acc: 0.598000
(Iteration 3601 / 9800) loss: 0.309518
(Epoch 37 / 100) train acc: 0.981000; val_acc: 0.596000
(Iteration 3701 / 9800) loss: 0.271904
(Epoch 38 / 100) train acc: 0.984000; val_acc: 0.597000
(Iteration 3801 / 9800) loss: 0.221919
(Epoch 39 / 100) train acc: 0.980000; val_acc: 0.610000
(Iteration 3901 / 9800) loss: 0.247744
(Epoch 40 / 100) train acc: 0.990000; val_acc: 0.601000
(Iteration 4001 / 9800) loss: 0.211057
(Epoch 41 / 100) train acc: 0.986000; val_acc: 0.602000
(Iteration 4101 / 9800) loss: 0.215980
(Epoch 42 / 100) train acc: 0.989000; val_acc: 0.602000
(Iteration 4201 / 9800) loss: 0.189695
(Epoch 43 / 100) train acc: 0.990000; val_acc: 0.600000
(Iteration 4301 / 9800) loss: 0.237263
(Epoch 44 / 100) train acc: 0.989000; val_acc: 0.593000
(Iteration 4401 / 9800) loss: 0.149792
(Epoch 45 / 100) train acc: 0.981000; val_acc: 0.603000
(Iteration 4501 / 9800) loss: 0.195684
(Epoch 46 / 100) train acc: 0.994000; val_acc: 0.605000
(Iteration 4601 / 9800) loss: 0.225804
(Epoch 47 / 100) train acc: 0.990000; val_acc: 0.608000
(Iteration 4701 / 9800) loss: 0.227298
(Epoch 48 / 100) train acc: 0.993000; val_acc: 0.604000
(Iteration 4801 / 9800) loss: 0.155038
(Epoch 49 / 100) train acc: 0.997000; val_acc: 0.601000
(Epoch 50 / 100) train acc: 0.995000; val_acc: 0.611000
(Iteration 4901 / 9800) loss: 0.179200
(Epoch 51 / 100) train acc: 0.994000; val_acc: 0.603000
(Iteration 5001 / 9800) loss: 0.148308
(Epoch 52 / 100) train acc: 0.996000; val_acc: 0.601000
(Iteration 5101 / 9800) loss: 0.203314
(Epoch 53 / 100) train acc: 0.998000; val_acc: 0.602000
(Iteration 5201 / 9800) loss: 0.133153
(Epoch 54 / 100) train acc: 0.998000; val_acc: 0.613000
(Iteration 5301 / 9800) loss: 0.179897
(Epoch 55 / 100) train acc: 0.997000; val_acc: 0.603000
(Iteration 5401 / 9800) loss: 0.138111
(Epoch 56 / 100) train acc: 0.998000; val_acc: 0.606000
(Iteration 5501 / 9800) loss: 0.120119
(Epoch 57 / 100) train acc: 0.999000; val_acc: 0.607000
(Iteration 5601 / 9800) loss: 0.167840
(Epoch 58 / 100) train acc: 0.996000; val_acc: 0.610000
(Iteration 5701 / 9800) loss: 0.188360
(Epoch 59 / 100) train acc: 0.996000; val_acc: 0.603000
(Iteration 5801 / 9800) loss: 0.190754
(Epoch 60 / 100) train acc: 0.999000; val_acc: 0.612000
(Iteration 5901 / 9800) loss: 0.189417
(Epoch 61 / 100) train acc: 0.998000; val_acc: 0.603000
(Iteration 6001 / 9800) loss: 0.206282
(Epoch 62 / 100) train acc: 0.997000; val_acc: 0.597000
(Iteration 6101 / 9800) loss: 0.140797
(Epoch 63 / 100) train acc: 0.997000; val_acc: 0.604000
(Iteration 6201 / 9800) loss: 0.153129
(Epoch 64 / 100) train acc: 0.999000; val_acc: 0.613000
(Iteration 6301 / 9800) loss: 0.166696
(Epoch 65 / 100) train acc: 0.997000; val_acc: 0.605000
(Iteration 6401 / 9800) loss: 0.113103
(Epoch 66 / 100) train acc: 0.999000; val_acc: 0.604000
```

```
(Iteration 6501 / 9800) loss: 0.103705
(Epoch 67 / 100) train acc: 0.998000; val_acc: 0.613000
(Iteration 6601 / 9800) loss: 0.100708
(Epoch 68 / 100) train acc: 0.999000; val_acc: 0.607000
(Iteration 6701 / 9800) loss: 0.108514
(Epoch 69 / 100) train acc: 0.999000; val_acc: 0.608000
(Iteration 6801 / 9800) loss: 0.097242
(Epoch 70 / 100) train acc: 0.999000; val_acc: 0.608000
(Iteration 6901 / 9800) loss: 0.119275
(Epoch 71 / 100) train acc: 0.999000; val_acc: 0.606000
(Iteration 7001 / 9800) loss: 0.092603
(Epoch 72 / 100) train acc: 0.999000; val_acc: 0.612000
(Iteration 7101 / 9800) loss: 0.140257
(Epoch 73 / 100) train acc: 0.999000; val_acc: 0.611000
(Iteration 7201 / 9800) loss: 0.101904
(Epoch 74 / 100) train acc: 0.999000; val_acc: 0.614000
(Iteration 7301 / 9800) loss: 0.109674
(Epoch 75 / 100) train acc: 0.999000; val_acc: 0.611000
(Iteration 7401 / 9800) loss: 0.111727
(Epoch 76 / 100) train acc: 0.998000; val_acc: 0.615000
(Iteration 7501 / 9800) loss: 0.128696
(Epoch 77 / 100) train acc: 1.000000; val_acc: 0.613000
(Iteration 7601 / 9800) loss: 0.145574
(Epoch 78 / 100) train acc: 0.999000; val_acc: 0.607000
(Iteration 7701 / 9800) loss: 0.134654
(Epoch 79 / 100) train acc: 0.997000; val_acc: 0.604000
(Iteration 7801 / 9800) loss: 0.114184
(Epoch 80 / 100) train acc: 0.999000; val_acc: 0.612000
(Iteration 7901 / 9800) loss: 0.173852
(Epoch 81 / 100) train acc: 0.999000; val_acc: 0.613000
(Iteration 8001 / 9800) loss: 0.126730
(Epoch 82 / 100) train acc: 1.000000; val_acc: 0.612000
(Iteration 8101 / 9800) loss: 0.085527
(Epoch 83 / 100) train acc: 0.997000; val_acc: 0.613000
(Iteration 8201 / 9800) loss: 0.090590
(Epoch 84 / 100) train acc: 0.999000; val_acc: 0.613000
(Iteration 8301 / 9800) loss: 0.109501
(Epoch 85 / 100) train acc: 0.999000; val_acc: 0.616000
(Iteration 8401 / 9800) loss: 0.103534
(Epoch 86 / 100) train acc: 0.999000; val_acc: 0.613000
(Iteration 8501 / 9800) loss: 0.096822
(Epoch 87 / 100) train acc: 1.000000; val_acc: 0.615000
(Iteration 8601 / 9800) loss: 0.095079
(Epoch 88 / 100) train acc: 0.998000; val_acc: 0.614000
(Iteration 8701 / 9800) loss: 0.125121
(Epoch 89 / 100) train acc: 0.999000; val_acc: 0.611000
(Iteration 8801 / 9800) loss: 0.127577
(Epoch 90 / 100) train acc: 0.999000; val_acc: 0.614000
(Iteration 8901 / 9800) loss: 0.101643
(Epoch 91 / 100) train acc: 0.999000; val_acc: 0.619000
(Iteration 9001 / 9800) loss: 0.125605
(Epoch 92 / 100) train acc: 0.999000; val_acc: 0.611000
(Iteration 9101 / 9800) loss: 0.133782
(Epoch 93 / 100) train acc: 0.999000; val_acc: 0.609000
(Iteration 9201 / 9800) loss: 0.095340
(Epoch 94 / 100) train acc: 0.999000; val_acc: 0.615000
(Iteration 9301 / 9800) loss: 0.129781
(Epoch 95 / 100) train acc: 0.997000; val_acc: 0.614000
(Iteration 9401 / 9800) loss: 0.121475
(Epoch 96 / 100) train acc: 0.999000; val_acc: 0.612000
(Iteration 9501 / 9800) loss: 0.133388
(Epoch 97 / 100) train acc: 0.999000; val_acc: 0.615000
(Iteration 9601 / 9800) loss: 0.118243
(Epoch 98 / 100) train acc: 0.998000; val_acc: 0.615000
(Iteration 9701 / 9800) loss: 0.100336
(Epoch 99 / 100) train acc: 1.000000; val_acc: 0.611000
```

```
(Epoch 100 / 100) train acc: 1.000000; val_acc: 0.613000

Test set accuracy: 0.593
Validation set accuracy: 0.615
```