Dropout

February 9, 2022

1 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`.

[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))))
```

[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 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.

[12]:
```
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:  10.0029854848449
Mean of train-time output:  9.995745723135094
Mean of test-time output:  10.0029854848449
Fraction of train-time output set to zero:  0.300448
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.0029854848449
Mean of train-time output:  9.97811571450516
Mean of test-time output:  10.0029854848449
Fraction of train-time output set to zero:  0.60082
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.0029854848449
Mean of train-time output:  9.999767820258494
Mean of test-time output:  10.0029854848449
Fraction of train-time output set to zero:  0.750092
Fraction of test-time output set to zero:  0.0
```

1.2 Dropout backward pass

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

[13]:
```
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], x, dout)

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

```
dx relative error:  1.892907914163588e-11
```

1.3 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).

[18]:
```
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:  2.303043161170242
W1 relative error: 4.795196913586914e-07
W2 relative error: 1.9717710552624993e-07
W3 relative error: 1.5587099391417707e-07
b1 relative error: 2.0336164373878067e-08
b2 relative error: 1.686315567518667e-09
b3 relative error: 1.1144421861081857e-10


Running check with dropout =  0.25
Initial loss:  2.3057748188351503
W1 relative error: 1.8156244550170462e-07
W2 relative error: 1.4561937925755805e-06
W3 relative error: 1.8547542000412227e-07
b1 relative error: 1.7241183361982744e-08
b2 relative error: 3.342987442194132e-09
b3 relative error: 1.8619747824927798e-10


Running check with dropout =  0.5
Initial loss:  2.2994400290646637
W1 relative error: 3.3884155778064 82e-08
W2 relative error: 8.884396392344106e-08
W3 relative error: 2.3767592608064155e-08
b1 relative error: 9.935223182732403e-10
b2 relative error: 8.032917281149487e-10
b3 relative error: 1.4333807510530193e-10
```

1.4 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.

[19]:
```
# 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: 2.298716
(Epoch 0 / 25) train acc: 0.132000; val_acc: 0.146000
(Epoch 1 / 25) train acc: 0.118000; val_acc: 0.131000
```

```
(Epoch 2 / 25) train acc: 0.220000; val_acc: 0.214000
(Epoch 3 / 25) train acc: 0.206000; val_acc: 0.180000
(Epoch 4 / 25) train acc: 0.220000; val_acc: 0.193000
(Epoch 5 / 25) train acc: 0.264000; val_acc: 0.229000
(Epoch 6 / 25) train acc: 0.268000; val_acc: 0.203000
(Epoch 7 / 25) train acc: 0.266000; val_acc: 0.212000
(Epoch 8 / 25) train acc: 0.282000; val_acc: 0.236000
(Epoch 9 / 25) train acc: 0.310000; val_acc: 0.255000
(Epoch 10 / 25) train acc: 0.320000; val_acc: 0.267000
(Epoch 11 / 25) train acc: 0.338000; val_acc: 0.273000
(Epoch 12 / 25) train acc: 0.346000; val_acc: 0.278000
(Epoch 13 / 25) train acc: 0.332000; val_acc: 0.279000
(Epoch 14 / 25) train acc: 0.328000; val_acc: 0.284000
(Epoch 15 / 25) train acc: 0.354000; val_acc: 0.271000
(Epoch 16 / 25) train acc: 0.386000; val_acc: 0.277000
(Epoch 17 / 25) train acc: 0.388000; val_acc: 0.297000
(Epoch 18 / 25) train acc: 0.402000; val_acc: 0.280000
(Epoch 19 / 25) train acc: 0.388000; val_acc: 0.274000
(Epoch 20 / 25) train acc: 0.386000; val_acc: 0.274000
(Iteration 101 / 125) loss: 1.919649
(Epoch 21 / 25) train acc: 0.402000; val_acc: 0.272000
(Epoch 22 / 25) train acc: 0.440000; val_acc: 0.286000
(Epoch 23 / 25) train acc: 0.458000; val_acc: 0.295000
(Epoch 24 / 25) train acc: 0.462000; val_acc: 0.311000
(Epoch 25 / 25) train acc: 0.466000; val_acc: 0.297000
```

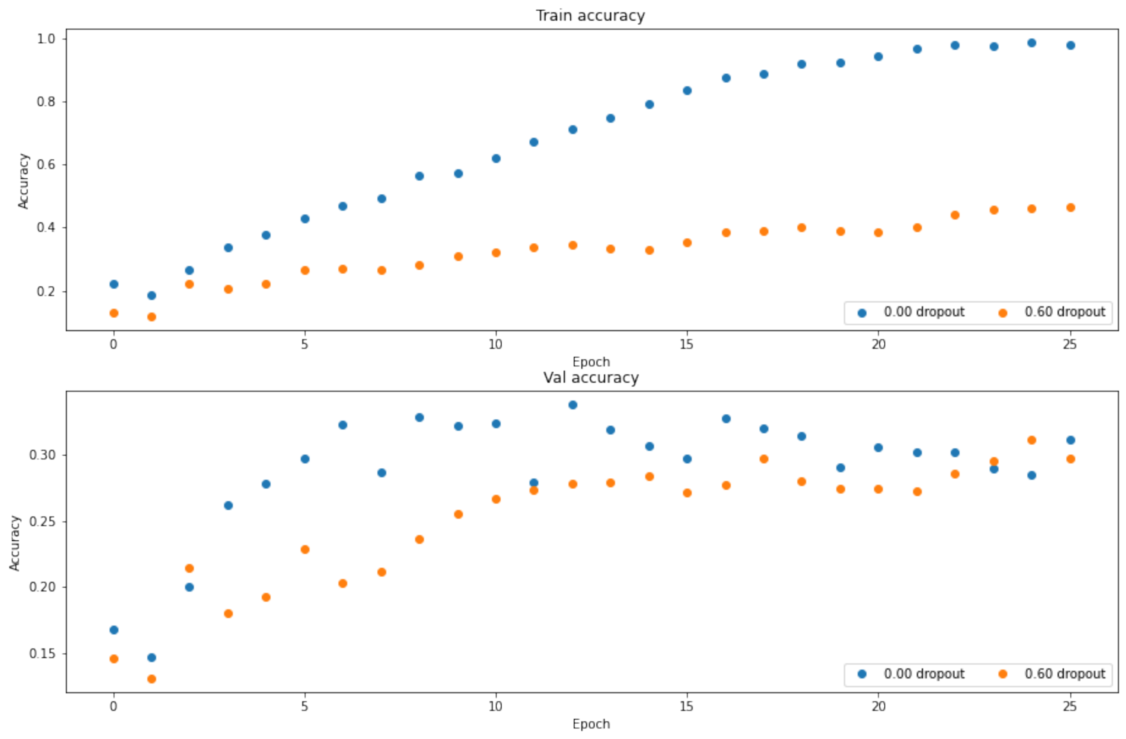
[20]:
```
# Plot train and validation accuracies of the two models

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()
```



1.5 Question

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

1.6 Answer:

Dropout is providing a regularizing effect by limiting overfitting to the training data The first plot clearly indicates that the network without dropout fits the training data much better with higher accuracy. But when we try to generalize to the validation set, we see the performance is quite comparable at the end. Thus droput is limiting overfitting the training data with no sacrifice to the validation accuracy (which is the accuracy we care about).

1.7 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.

[31]:
```python
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #

# set parameters
hidden_dims = [1024, 1024, 1024, 1024]
learning_rate = 1e-3
weight_scale = 0.01
lr_decay = 0.90
dropout = 0.65
update_rule = 'adam'
# create FullyConnectedNet
model = FullyConnectedNet(hidden_dims=hidden_dims, weight_scale=weight_scale,␣
 ↪dropout=dropout, use_batchnorm=True, reg=0.0)
# solve
solver = Solver(model, data,
                num_epochs=40, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 5e-4,
                },
                lr_decay=lr_decay,
                verbose=True, print_every=100)
solver.train()
# print out the validation accuracy
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('Validation set accuracy: {}'.format(np.mean(y_val_pred ==␣
 ↪data['y_val'])))
print('Test set accuracy: {}'.format(np.mean(y_test_pred == data['y_test'])))

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

```
(Iteration 1 / 19600) loss: 2.319197
(Epoch 0 / 40) train acc: 0.120000; val_acc: 0.124000
(Iteration 101 / 19600) loss: 1.963282
```

```
(Iteration 201 / 19600) loss: 1.886046
(Iteration 301 / 19600) loss: 1.877358
(Iteration 401 / 19600) loss: 1.705007
(Epoch 1 / 40) train acc: 0.428000; val_acc: 0.431000
(Iteration 501 / 19600) loss: 1.697679
(Iteration 601 / 19600) loss: 1.610060
(Iteration 701 / 19600) loss: 1.843761
(Iteration 801 / 19600) loss: 1.687082
(Iteration 901 / 19600) loss: 1.559826
(Epoch 2 / 40) train acc: 0.443000; val_acc: 0.463000
(Iteration 1001 / 19600) loss: 1.512483
(Iteration 1101 / 19600) loss: 1.535922
(Iteration 1201 / 19600) loss: 1.573896
(Iteration 1301 / 19600) loss: 1.504816
(Iteration 1401 / 19600) loss: 1.450323
(Epoch 3 / 40) train acc: 0.498000; val_acc: 0.474000
(Iteration 1501 / 19600) loss: 1.442316
(Iteration 1601 / 19600) loss: 1.623230
(Iteration 1701 / 19600) loss: 1.601297
(Iteration 1801 / 19600) loss: 1.588590
(Iteration 1901 / 19600) loss: 1.585946
(Epoch 4 / 40) train acc: 0.525000; val_acc: 0.502000
(Iteration 2001 / 19600) loss: 1.468565
(Iteration 2101 / 19600) loss: 1.460408
(Iteration 2201 / 19600) loss: 1.562722
(Iteration 2301 / 19600) loss: 1.553577
(Iteration 2401 / 19600) loss: 1.355722
(Epoch 5 / 40) train acc: 0.485000; val_acc: 0.505000
(Iteration 2501 / 19600) loss: 1.771677
(Iteration 2601 / 19600) loss: 1.451407
(Iteration 2701 / 19600) loss: 1.555366
(Iteration 2801 / 19600) loss: 1.378734
(Iteration 2901 / 19600) loss: 1.408954
(Epoch 6 / 40) train acc: 0.504000; val_acc: 0.506000
(Iteration 3001 / 19600) loss: 1.609983
(Iteration 3101 / 19600) loss: 1.434493
(Iteration 3201 / 19600) loss: 1.410258
(Iteration 3301 / 19600) loss: 1.499204
(Iteration 3401 / 19600) loss: 1.525788
(Epoch 7 / 40) train acc: 0.539000; val_acc: 0.520000
(Iteration 3501 / 19600) loss: 1.332758
(Iteration 3601 / 19600) loss: 1.541873
(Iteration 3701 / 19600) loss: 1.401716
(Iteration 3801 / 19600) loss: 1.300585
(Iteration 3901 / 19600) loss: 1.278040
(Epoch 8 / 40) train acc: 0.560000; val_acc: 0.521000
(Iteration 4001 / 19600) loss: 1.470663
(Iteration 4101 / 19600) loss: 1.413896
```

```
(Iteration 4201 / 19600) loss: 1.391558
(Iteration 4301 / 19600) loss: 1.408812
(Iteration 4401 / 19600) loss: 1.498123
(Epoch 9 / 40) train acc: 0.579000; val_acc: 0.534000
(Iteration 4501 / 19600) loss: 1.213835
(Iteration 4601 / 19600) loss: 1.302080
(Iteration 4701 / 19600) loss: 1.463121
(Iteration 4801 / 19600) loss: 1.351313
(Epoch 10 / 40) train acc: 0.561000; val_acc: 0.542000
(Iteration 4901 / 19600) loss: 1.173530
(Iteration 5001 / 19600) loss: 1.388805
(Iteration 5101 / 19600) loss: 1.308384
(Iteration 5201 / 19600) loss: 1.132627
(Iteration 5301 / 19600) loss: 1.434476
(Epoch 11 / 40) train acc: 0.586000; val_acc: 0.547000
(Iteration 5401 / 19600) loss: 1.497799
(Iteration 5501 / 19600) loss: 1.353251
(Iteration 5601 / 19600) loss: 1.309968
(Iteration 5701 / 19600) loss: 1.360504
(Iteration 5801 / 19600) loss: 1.450646
(Epoch 12 / 40) train acc: 0.583000; val_acc: 0.546000
(Iteration 5901 / 19600) loss: 1.336092
(Iteration 6001 / 19600) loss: 1.143651
(Iteration 6101 / 19600) loss: 1.341872
(Iteration 6201 / 19600) loss: 1.354553
(Iteration 6301 / 19600) loss: 1.293437
(Epoch 13 / 40) train acc: 0.574000; val_acc: 0.546000
(Iteration 6401 / 19600) loss: 1.234187
(Iteration 6501 / 19600) loss: 1.323418
(Iteration 6601 / 19600) loss: 1.317785
(Iteration 6701 / 19600) loss: 1.506635
(Iteration 6801 / 19600) loss: 1.086035
(Epoch 14 / 40) train acc: 0.620000; val_acc: 0.554000
(Iteration 6901 / 19600) loss: 1.362260
(Iteration 7001 / 19600) loss: 1.386244
(Iteration 7101 / 19600) loss: 1.281761
(Iteration 7201 / 19600) loss: 1.273431
(Iteration 7301 / 19600) loss: 1.349138
(Epoch 15 / 40) train acc: 0.598000; val_acc: 0.552000
(Iteration 7401 / 19600) loss: 1.279325
(Iteration 7501 / 19600) loss: 1.400798
(Iteration 7601 / 19600) loss: 1.468715
(Iteration 7701 / 19600) loss: 1.184215
(Iteration 7801 / 19600) loss: 1.483129
(Epoch 16 / 40) train acc: 0.616000; val_acc: 0.545000
(Iteration 7901 / 19600) loss: 1.375803
(Iteration 8001 / 19600) loss: 1.095799
(Iteration 8101 / 19600) loss: 1.375200
```

```
(Iteration 8201 / 19600) loss: 1.559873
(Iteration 8301 / 19600) loss: 1.167754
(Epoch 17 / 40) train acc: 0.641000; val_acc: 0.540000
(Iteration 8401 / 19600) loss: 1.228685
(Iteration 8501 / 19600) loss: 1.131828
(Iteration 8601 / 19600) loss: 1.241578
(Iteration 8701 / 19600) loss: 1.120189
(Iteration 8801 / 19600) loss: 1.303789
(Epoch 18 / 40) train acc: 0.625000; val_acc: 0.556000
(Iteration 8901 / 19600) loss: 1.178950
(Iteration 9001 / 19600) loss: 1.305805
(Iteration 9101 / 19600) loss: 1.135359
(Iteration 9201 / 19600) loss: 1.330246
(Iteration 9301 / 19600) loss: 1.181554
(Epoch 19 / 40) train acc: 0.625000; val_acc: 0.554000
(Iteration 9401 / 19600) loss: 1.179319
(Iteration 9501 / 19600) loss: 1.262856
(Iteration 9601 / 19600) loss: 1.259469
(Iteration 9701 / 19600) loss: 1.162605
(Epoch 20 / 40) train acc: 0.633000; val_acc: 0.545000
(Iteration 9801 / 19600) loss: 1.197180
(Iteration 9901 / 19600) loss: 1.192975
(Iteration 10001 / 19600) loss: 1.527332
(Iteration 10101 / 19600) loss: 1.313705
(Iteration 10201 / 19600) loss: 1.392250
(Epoch 21 / 40) train acc: 0.673000; val_acc: 0.558000
(Iteration 10301 / 19600) loss: 1.471109
(Iteration 10401 / 19600) loss: 1.175193
(Iteration 10501 / 19600) loss: 1.468333
(Iteration 10601 / 19600) loss: 1.414879
(Iteration 10701 / 19600) loss: 1.253829
(Epoch 22 / 40) train acc: 0.632000; val_acc: 0.561000
(Iteration 10801 / 19600) loss: 1.110335
(Iteration 10901 / 19600) loss: 1.234992
(Iteration 11001 / 19600) loss: 1.150242
(Iteration 11101 / 19600) loss: 1.174087
(Iteration 11201 / 19600) loss: 1.344736
(Epoch 23 / 40) train acc: 0.639000; val_acc: 0.564000
(Iteration 11301 / 19600) loss: 1.024447
(Iteration 11401 / 19600) loss: 1.225536
(Iteration 11501 / 19600) loss: 1.228083
(Iteration 11601 / 19600) loss: 1.368532
(Iteration 11701 / 19600) loss: 1.196327
(Epoch 24 / 40) train acc: 0.672000; val_acc: 0.554000
(Iteration 11801 / 19600) loss: 1.172113
(Iteration 11901 / 19600) loss: 1.288999
(Iteration 12001 / 19600) loss: 1.078615
(Iteration 12101 / 19600) loss: 1.235464
```

```
(Iteration 12201 / 19600) loss: 1.408260
(Epoch 25 / 40) train acc: 0.625000; val_acc: 0.557000
(Iteration 12301 / 19600) loss: 1.186526
(Iteration 12401 / 19600) loss: 1.225902
(Iteration 12501 / 19600) loss: 1.249146
(Iteration 12601 / 19600) loss: 1.284568
(Iteration 12701 / 19600) loss: 1.169589
(Epoch 26 / 40) train acc: 0.649000; val_acc: 0.562000
(Iteration 12801 / 19600) loss: 1.202635
(Iteration 12901 / 19600) loss: 1.259158
(Iteration 13001 / 19600) loss: 1.064012
(Iteration 13101 / 19600) loss: 1.164635
(Iteration 13201 / 19600) loss: 1.177146
(Epoch 27 / 40) train acc: 0.621000; val_acc: 0.568000
(Iteration 13301 / 19600) loss: 1.133549
(Iteration 13401 / 19600) loss: 1.242085
(Iteration 13501 / 19600) loss: 1.300241
(Iteration 13601 / 19600) loss: 1.237024
(Iteration 13701 / 19600) loss: 1.212179
(Epoch 28 / 40) train acc: 0.622000; val_acc: 0.563000
(Iteration 13801 / 19600) loss: 1.255180
(Iteration 13901 / 19600) loss: 1.032810
(Iteration 14001 / 19600) loss: 1.292151
(Iteration 14101 / 19600) loss: 1.183742
(Iteration 14201 / 19600) loss: 1.230419
(Epoch 29 / 40) train acc: 0.673000; val_acc: 0.560000
(Iteration 14301 / 19600) loss: 1.138439
(Iteration 14401 / 19600) loss: 1.175002
(Iteration 14501 / 19600) loss: 1.189359
(Iteration 14601 / 19600) loss: 1.328503
(Epoch 30 / 40) train acc: 0.638000; val_acc: 0.563000
(Iteration 14701 / 19600) loss: 1.212837
(Iteration 14801 / 19600) loss: 1.372738
(Iteration 14901 / 19600) loss: 1.217731
(Iteration 15001 / 19600) loss: 0.938929
(Iteration 15101 / 19600) loss: 1.211556
(Epoch 31 / 40) train acc: 0.649000; val_acc: 0.570000
(Iteration 15201 / 19600) loss: 1.152620
(Iteration 15301 / 19600) loss: 1.365274
(Iteration 15401 / 19600) loss: 1.203772
(Iteration 15501 / 19600) loss: 1.275052
(Iteration 15601 / 19600) loss: 1.039293
(Epoch 32 / 40) train acc: 0.651000; val_acc: 0.571000
(Iteration 15701 / 19600) loss: 1.306846
(Iteration 15801 / 19600) loss: 1.088925
(Iteration 15901 / 19600) loss: 1.160546
(Iteration 16001 / 19600) loss: 1.157342
(Iteration 16101 / 19600) loss: 1.124629
```

```
(Epoch 33 / 40) train acc: 0.649000; val_acc: 0.566000
(Iteration 16201 / 19600) loss: 1.242696
(Iteration 16301 / 19600) loss: 1.151341
(Iteration 16401 / 19600) loss: 1.284520
(Iteration 16501 / 19600) loss: 1.289337
(Iteration 16601 / 19600) loss: 1.256997
(Epoch 34 / 40) train acc: 0.642000; val_acc: 0.571000
(Iteration 16701 / 19600) loss: 1.159255
(Iteration 16801 / 19600) loss: 1.190853
(Iteration 16901 / 19600) loss: 1.037460
(Iteration 17001 / 19600) loss: 1.293466
(Iteration 17101 / 19600) loss: 1.204287
(Epoch 35 / 40) train acc: 0.670000; val_acc: 0.571000
(Iteration 17201 / 19600) loss: 1.330501
(Iteration 17301 / 19600) loss: 1.046763
(Iteration 17401 / 19600) loss: 1.289613
(Iteration 17501 / 19600) loss: 1.150291
(Iteration 17601 / 19600) loss: 1.125224
(Epoch 36 / 40) train acc: 0.665000; val_acc: 0.572000
(Iteration 17701 / 19600) loss: 1.189774
(Iteration 17801 / 19600) loss: 1.193009
(Iteration 17901 / 19600) loss: 1.185220
(Iteration 18001 / 19600) loss: 1.198032
(Iteration 18101 / 19600) loss: 1.181664
(Epoch 37 / 40) train acc: 0.637000; val_acc: 0.576000
(Iteration 18201 / 19600) loss: 1.279130
(Iteration 18301 / 19600) loss: 1.150239
(Iteration 18401 / 19600) loss: 1.178781
(Iteration 18501 / 19600) loss: 1.210802
(Iteration 18601 / 19600) loss: 1.207838
(Epoch 38 / 40) train acc: 0.658000; val_acc: 0.569000
(Iteration 18701 / 19600) loss: 1.278023
(Iteration 18801 / 19600) loss: 1.138024
(Iteration 18901 / 19600) loss: 1.214834
(Iteration 19001 / 19600) loss: 1.180616
(Iteration 19101 / 19600) loss: 1.158760
(Epoch 39 / 40) train acc: 0.657000; val_acc: 0.570000
(Iteration 19201 / 19600) loss: 1.364472
(Iteration 19301 / 19600) loss: 1.265520
(Iteration 19401 / 19600) loss: 1.212633
(Iteration 19501 / 19600) loss: 1.128712
(Epoch 40 / 40) train acc: 0.669000; val_acc: 0.571000
Validation set accuracy: 0.605
Test set accuracy: 0.575
```

[32]:
```
print('Validation set accuracy: {}'.format(np.mean(y_val_pred ==␣
↪data['y_val'])))
```

```
Validation set accuracy: 0.605
```