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_numeri
cal_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
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)
X_test: (1000, 3, 32, 32)
y_val: (1000,)
X_val: (1000, 3, 32, 32)
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 [4]:
```
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).me
an())
    print('Fraction of test-time output set to zero: ', (out_test == 0
).mean())
```

```
Running tests with p =  0.3
Mean of input:  9.99853206456
Mean of train-time output:  10.0156191475
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.699472
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.99853206456
Mean of train-time output:  10.0040205996
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.399612
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.99853206456
Mean of train-time output:  10.00547198
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.249576
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 [6]: 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:  1.89289661024e-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 [10]:
```
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 [0.5, 0.75, 1.0]:
    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], verb
ose=False, h=1e-5)
        print('{} relative error: {}'.format(name, rel_error(grad_num,
grads[name])))
    print('\n')
```

```
Running check with dropout =  0.5
Initial loss:  2.30355723855
W1 relative error: 4.566234110910478e-08
W2 relative error: 4.395471174488098e-08
W3 relative error: 6.244950801146958e-08
b1 relative error: 2.0413189453171298e-09
b2 relative error: 2.785132687644587e-09
b3 relative error: 1.3584809600954945e-10


Running check with dropout =  0.75
Initial loss:  2.30149284277
W1 relative error: 9.83241315204456e-08
W2 relative error: 1.2484064610109774e-07
W3 relative error: 7.072876331070812e-08
b1 relative error: 6.57071632012831e-08
b2 relative error: 2.4973364074125794e-09
b3 relative error: 1.5893216402745932e-10


Running check with dropout =  1.0
Initial loss:  2.30273025736
W1 relative error: 1.3610160941555372e-07
W2 relative error: 1.1681571878793191e-07
W3 relative error: 9.422106252896962e-08
b1 relative error: 1.1796170512665718e-08
b2 relative error: 2.062339792715741e-09
b3 relative error: 9.274709761196104e-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 [11]:
```
# 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 = [0.6, 1.0]
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: 2.300199
(Epoch 0 / 25) train acc: 0.158000; val_acc: 0.127000
(Epoch 1 / 25) train acc: 0.132000; val_acc: 0.121000
(Epoch 2 / 25) train acc: 0.204000; val_acc: 0.170000
(Epoch 3 / 25) train acc: 0.240000; val_acc: 0.192000
(Epoch 4 / 25) train acc: 0.312000; val_acc: 0.274000
(Epoch 5 / 25) train acc: 0.314000; val_acc: 0.269000
(Epoch 6 / 25) train acc: 0.364000; val_acc: 0.252000
(Epoch 7 / 25) train acc: 0.390000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.386000; val_acc: 0.290000
(Epoch 9 / 25) train acc: 0.372000; val_acc: 0.267000
(Epoch 10 / 25) train acc: 0.424000; val_acc: 0.286000
(Epoch 11 / 25) train acc: 0.396000; val_acc: 0.275000
(Epoch 12 / 25) train acc: 0.458000; val_acc: 0.299000
(Epoch 13 / 25) train acc: 0.496000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.492000; val_acc: 0.299000
(Epoch 15 / 25) train acc: 0.550000; val_acc: 0.296000
(Epoch 16 / 25) train acc: 0.584000; val_acc: 0.297000
(Epoch 17 / 25) train acc: 0.582000; val_acc: 0.309000
(Epoch 18 / 25) train acc: 0.612000; val_acc: 0.306000
(Epoch 19 / 25) train acc: 0.628000; val_acc: 0.323000
(Epoch 20 / 25) train acc: 0.608000; val_acc: 0.324000
(Iteration 101 / 125) loss: 1.369535
(Epoch 21 / 25) train acc: 0.644000; val_acc: 0.330000
(Epoch 22 / 25) train acc: 0.708000; val_acc: 0.341000
(Epoch 23 / 25) train acc: 0.690000; val_acc: 0.297000
(Epoch 24 / 25) train acc: 0.740000; val_acc: 0.300000
(Epoch 25 / 25) train acc: 0.750000; val_acc: 0.329000
(Iteration 1 / 125) loss: 2.300607
(Epoch 0 / 25) train acc: 0.172000; val_acc: 0.167000
(Epoch 1 / 25) train acc: 0.210000; val_acc: 0.197000
(Epoch 2 / 25) train acc: 0.284000; val_acc: 0.240000
(Epoch 3 / 25) train acc: 0.302000; val_acc: 0.246000
(Epoch 4 / 25) train acc: 0.392000; val_acc: 0.289000
(Epoch 5 / 25) train acc: 0.420000; val_acc: 0.274000
(Epoch 6 / 25) train acc: 0.420000; val_acc: 0.304000
(Epoch 7 / 25) train acc: 0.474000; val_acc: 0.293000
(Epoch 8 / 25) train acc: 0.516000; val_acc: 0.330000
(Epoch 9 / 25) train acc: 0.566000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.620000; val_acc: 0.321000
(Epoch 11 / 25) train acc: 0.656000; val_acc: 0.317000
(Epoch 12 / 25) train acc: 0.676000; val_acc: 0.319000
(Epoch 13 / 25) train acc: 0.680000; val_acc: 0.304000
(Epoch 14 / 25) train acc: 0.754000; val_acc: 0.321000
(Epoch 15 / 25) train acc: 0.800000; val_acc: 0.321000
(Epoch 16 / 25) train acc: 0.800000; val_acc: 0.299000
(Epoch 17 / 25) train acc: 0.870000; val_acc: 0.306000
(Epoch 18 / 25) train acc: 0.892000; val_acc: 0.300000
(Epoch 19 / 25) train acc: 0.904000; val_acc: 0.285000
(Epoch 20 / 25) train acc: 0.924000; val_acc: 0.316000
(Iteration 101 / 125) loss: 0.255556
(Epoch 21 / 25) train acc: 0.948000; val_acc: 0.285000
(Epoch 22 / 25) train acc: 0.954000; val_acc: 0.286000
(Epoch 23 / 25) train acc: 0.966000; val_acc: 0.306000
(Epoch 24 / 25) train acc: 0.970000; val_acc: 0.285000
(Epoch 25 / 25) train acc: 0.974000; val_acc: 0.287000
```

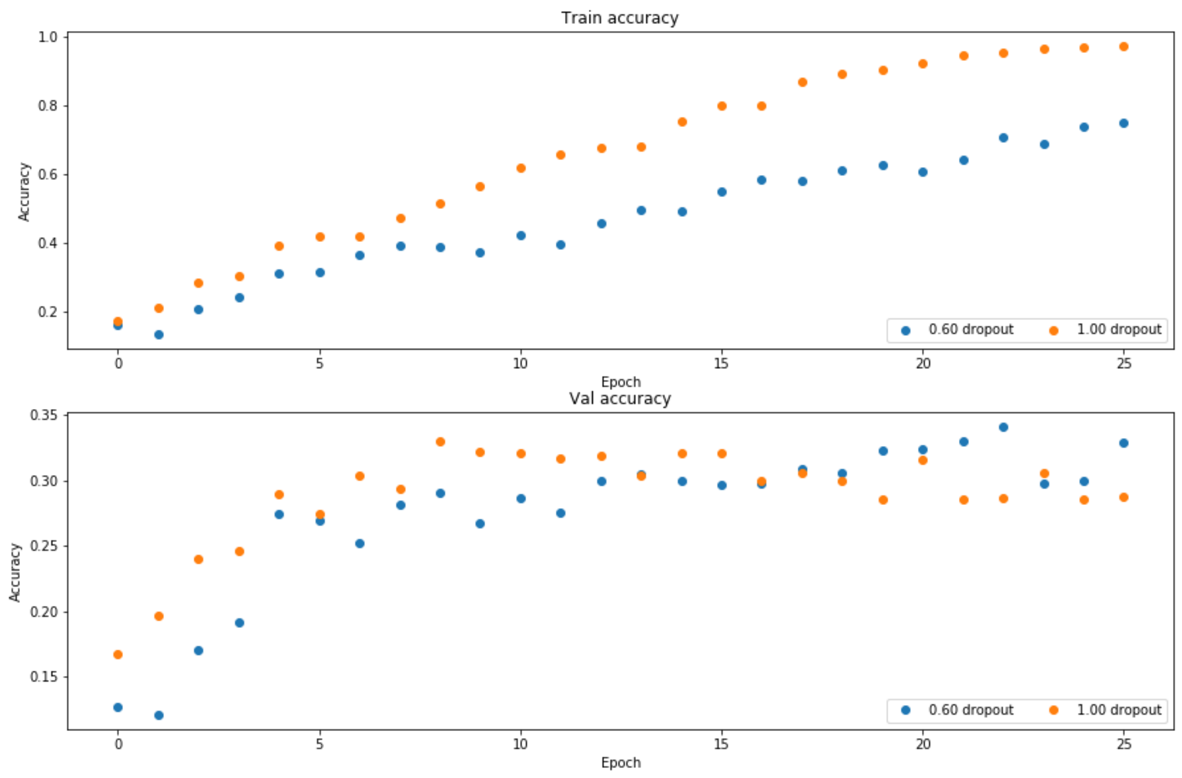
In [12]:
```
# Plot train and validation accuracies of the two models

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 drop
out' % 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 dropou
t' % 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. We can see that when we used dropout, the training accuracy was smaller than the accuracy of the model without dropout (which makes sense because the network with dropout is effectively with less neurons operating), but at the same time the validation accuracy remained roughly the same as the accuracy of the model that didn't incorporate dropout. Since the difference between the training and validation accuracy is bigger for the model with dropout, it means that dropout helped with the generalization of the model, which is a form of regularization.

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%)) / 23%, 1) where if you get 55% or higher validation accuracy, you get full points.

In [19]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #
optimizer = 'adam'
layer_dims = [800, 650, 650]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale,
                          use_batchnorm=True, dropout=0.5)

solver = Solver(model, data,
                num_epochs=10, batch_size=150,
                update_rule=optimizer,
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=lr_decay,
                verbose=True, print_every=50)
solver.train()

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

```
(Iteration 1 / 3260) loss: 2.309706
(Epoch 0 / 10) train acc: 0.197000; val_acc: 0.205000
(Iteration 51 / 3260) loss: 1.753303
(Iteration 101 / 3260) loss: 1.677952
(Iteration 151 / 3260) loss: 1.682492
(Iteration 201 / 3260) loss: 1.781695
(Iteration 251 / 3260) loss: 1.556207
(Iteration 301 / 3260) loss: 1.674324
(Epoch 1 / 10) train acc: 0.449000; val_acc: 0.471000
(Iteration 351 / 3260) loss: 1.666532
(Iteration 401 / 3260) loss: 1.574301
(Iteration 451 / 3260) loss: 1.462209
(Iteration 501 / 3260) loss: 1.506639
(Iteration 551 / 3260) loss: 1.467008
(Iteration 601 / 3260) loss: 1.592831
(Iteration 651 / 3260) loss: 1.517175
(Epoch 2 / 10) train acc: 0.522000; val_acc: 0.497000
(Iteration 701 / 3260) loss: 1.691948
(Iteration 751 / 3260) loss: 1.634745
(Iteration 801 / 3260) loss: 1.518882
(Iteration 851 / 3260) loss: 1.345086
(Iteration 901 / 3260) loss: 1.379469
(Iteration 951 / 3260) loss: 1.451329
(Epoch 3 / 10) train acc: 0.564000; val_acc: 0.507000
(Iteration 1001 / 3260) loss: 1.393822
(Iteration 1051 / 3260) loss: 1.323602
(Iteration 1101 / 3260) loss: 1.460633
(Iteration 1151 / 3260) loss: 1.457747
(Iteration 1201 / 3260) loss: 1.299865
(Iteration 1251 / 3260) loss: 1.294650
(Iteration 1301 / 3260) loss: 1.352543
(Epoch 4 / 10) train acc: 0.558000; val_acc: 0.526000
(Iteration 1351 / 3260) loss: 1.252901
(Iteration 1401 / 3260) loss: 1.276353
(Iteration 1451 / 3260) loss: 1.366664
(Iteration 1501 / 3260) loss: 1.298607
(Iteration 1551 / 3260) loss: 1.463464
(Iteration 1601 / 3260) loss: 1.429020
(Epoch 5 / 10) train acc: 0.538000; val_acc: 0.529000
(Iteration 1651 / 3260) loss: 1.417271
(Iteration 1701 / 3260) loss: 1.502363
(Iteration 1751 / 3260) loss: 1.194770
(Iteration 1801 / 3260) loss: 1.500889
(Iteration 1851 / 3260) loss: 1.297009
(Iteration 1901 / 3260) loss: 1.403091
(Iteration 1951 / 3260) loss: 1.318034
(Epoch 6 / 10) train acc: 0.582000; val_acc: 0.538000
(Iteration 2001 / 3260) loss: 1.275208
(Iteration 2051 / 3260) loss: 1.337344
(Iteration 2101 / 3260) loss: 1.348497
(Iteration 2151 / 3260) loss: 1.312881
(Iteration 2201 / 3260) loss: 1.283106
(Iteration 2251 / 3260) loss: 1.321507
(Epoch 7 / 10) train acc: 0.585000; val_acc: 0.535000
(Iteration 2301 / 3260) loss: 1.233513
(Iteration 2351 / 3260) loss: 1.255630
(Iteration 2401 / 3260) loss: 1.185922
```

```
(Iteration 2451 / 3260) loss: 1.326277
(Iteration 2501 / 3260) loss: 1.235982
(Iteration 2551 / 3260) loss: 1.212447
(Iteration 2601 / 3260) loss: 1.270914
(Epoch 8 / 10) train acc: 0.635000; val_acc: 0.555000
(Iteration 2651 / 3260) loss: 1.199987
(Iteration 2701 / 3260) loss: 1.346234
(Iteration 2751 / 3260) loss: 1.089702
(Iteration 2801 / 3260) loss: 1.153776
(Iteration 2851 / 3260) loss: 1.194224
(Iteration 2901 / 3260) loss: 1.269478
(Epoch 9 / 10) train acc: 0.609000; val_acc: 0.551000
(Iteration 2951 / 3260) loss: 1.030614
(Iteration 3001 / 3260) loss: 1.090293
(Iteration 3051 / 3260) loss: 1.232051
(Iteration 3101 / 3260) loss: 1.090456
(Iteration 3151 / 3260) loss: 1.244426
(Iteration 3201 / 3260) loss: 1.143223
(Iteration 3251 / 3260) loss: 1.317788
(Epoch 10 / 10) train acc: 0.629000; val_acc: 0.573000
```