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 a solid API for building these modular frameworks and training them, and we will use this very well implemented framework as opposed to "reinventing the wheel." This includes using the Solver, various utility functions, and the 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:  10.000075654537444
Mean of train-time output:  9.991849518681061
Mean of test-time output:  10.000075654537444
Fraction of train-time output set to zero:  0.700248
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.000075654537444
Mean of train-time output:  9.973702372669695
Mean of test-time output:  10.000075654537444
Fraction of train-time output set to zero:  0.401652
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.000075654537444
Mean of train-time output:  9.991893782187312
Mean of test-time output:  10.000075654537444
Fraction of train-time output set to zero:  0.25066
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], x, dout)

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

```
dx relative error:  5.4456027087706535e-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 [0.5, 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], verbose=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:  2.309771209610118
W1 relative error: 2.694274363733021e-07
W2 relative error: 7.439246147919978e-08
W3 relative error: 1.910371122296728e-08
b1 relative error: 4.112891126518e-09
b2 relative error: 5.756217724722137e-10
b3 relative error: 1.3204470857080166e-10


Running check with dropout =  0.75
Initial loss:  2.306133548427975
W1 relative error: 8.72986097970181e-08
W2 relative error: 2.9777307885797295e-07
W3 relative error: 1.8832780806174298e-08
b1 relative error: 5.379486003985169e-08
b2 relative error: 3.6529949080385546e-09
b3 relative error: 9.987242764516995e-11


Running check with dropout =  1.0
Initial loss:  2.3053332250963194
W1 relative error: 1.2744095365229032e-06
W2 relative error: 4.678743300473988e-07
W3 relative error: 4.331673892536035e-08
b1 relative error: 4.0853539035931665e-08
b2 relative error: 1.951342257912746e-09
b3 relative error: 9.387142701440351e-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 [9]:

```python
# 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: 2.300340
(Epoch 0 / 25) train acc: 0.224000; val_acc: 0.167000
(Epoch 1 / 25) train acc: 0.170000; val_acc: 0.150000
(Epoch 2 / 25) train acc: 0.318000; val_acc: 0.261000
(Epoch 3 / 25) train acc: 0.296000; val_acc: 0.213000
(Epoch 4 / 25) train acc: 0.326000; val_acc: 0.246000
(Epoch 5 / 25) train acc: 0.338000; val_acc: 0.266000
(Epoch 6 / 25) train acc: 0.380000; val_acc: 0.243000
(Epoch 7 / 25) train acc: 0.440000; val_acc: 0.285000
(Epoch 8 / 25) train acc: 0.446000; val_acc: 0.294000
(Epoch 9 / 25) train acc: 0.410000; val_acc: 0.263000
(Epoch 10 / 25) train acc: 0.414000; val_acc: 0.278000
(Epoch 11 / 25) train acc: 0.480000; val_acc: 0.266000
(Epoch 12 / 25) train acc: 0.492000; val_acc: 0.312000
(Epoch 13 / 25) train acc: 0.482000; val_acc: 0.306000
(Epoch 14 / 25) train acc: 0.528000; val_acc: 0.313000
(Epoch 15 / 25) train acc: 0.526000; val_acc: 0.313000
(Epoch 16 / 25) train acc: 0.538000; val_acc: 0.312000
(Epoch 17 / 25) train acc: 0.534000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.566000; val_acc: 0.307000
(Epoch 19 / 25) train acc: 0.546000; val_acc: 0.312000
(Epoch 20 / 25) train acc: 0.590000; val_acc: 0.305000
(Iteration 101 / 125) loss: 1.449814
(Epoch 21 / 25) train acc: 0.566000; val_acc: 0.299000
(Epoch 22 / 25) train acc: 0.578000; val_acc: 0.308000
(Epoch 23 / 25) train acc: 0.610000; val_acc: 0.324000
(Epoch 24 / 25) train acc: 0.622000; val_acc: 0.300000
(Epoch 25 / 25) train acc: 0.628000; val_acc: 0.307000
(Iteration 1 / 125) loss: 2.303964
(Epoch 0 / 25) train acc: 0.142000; val_acc: 0.125000
(Epoch 1 / 25) train acc: 0.222000; val_acc: 0.193000
(Epoch 2 / 25) train acc: 0.326000; val_acc: 0.266000
(Epoch 3 / 25) train acc: 0.362000; val_acc: 0.257000
(Epoch 4 / 25) train acc: 0.426000; val_acc: 0.273000
(Epoch 5 / 25) train acc: 0.500000; val_acc: 0.266000
(Epoch 6 / 25) train acc: 0.570000; val_acc: 0.266000
(Epoch 7 / 25) train acc: 0.646000; val_acc: 0.285000
(Epoch 8 / 25) train acc: 0.662000; val_acc: 0.303000
(Epoch 9 / 25) train acc: 0.678000; val_acc: 0.276000
(Epoch 10 / 25) train acc: 0.804000; val_acc: 0.275000
(Epoch 11 / 25) train acc: 0.814000; val_acc: 0.292000
(Epoch 12 / 25) train acc: 0.800000; val_acc: 0.277000
(Epoch 13 / 25) train acc: 0.846000; val_acc: 0.264000
(Epoch 14 / 25) train acc: 0.846000; val_acc: 0.286000
(Epoch 15 / 25) train acc: 0.916000; val_acc: 0.288000
(Epoch 16 / 25) train acc: 0.922000; val_acc: 0.291000
(Epoch 17 / 25) train acc: 0.946000; val_acc: 0.312000
(Epoch 18 / 25) train acc: 0.968000; val_acc: 0.291000
(Epoch 19 / 25) train acc: 0.980000; val_acc: 0.282000
(Epoch 20 / 25) train acc: 0.986000; val_acc: 0.289000
(Iteration 101 / 125) loss: 0.076472
(Epoch 21 / 25) train acc: 0.984000; val_acc: 0.281000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.289000
(Epoch 23 / 25) train acc: 0.990000; val_acc: 0.298000
(Epoch 24 / 25) train acc: 0.976000; val_acc: 0.268000
(Epoch 25 / 25) train acc: 0.996000; val_acc: 0.268000
```

In [10]:

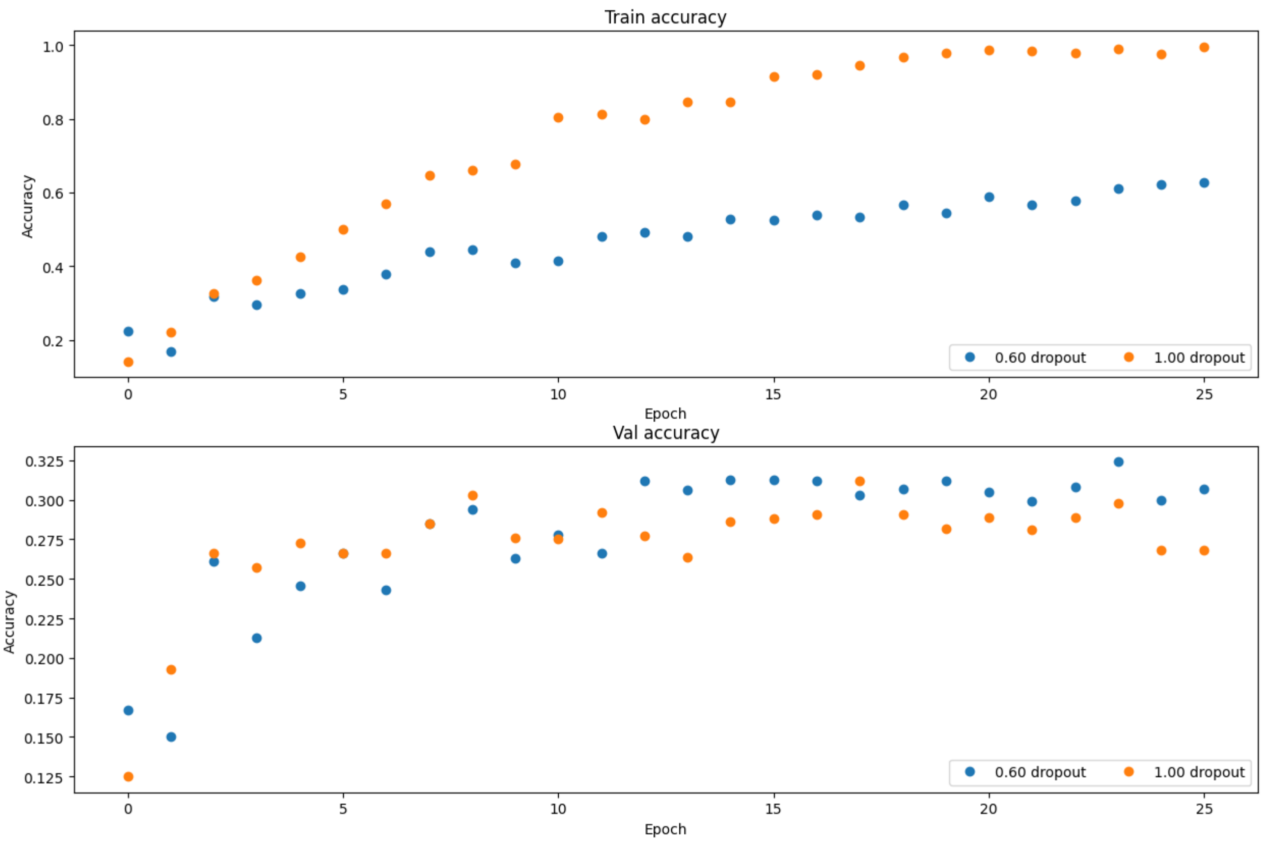
```
# Plot train and validation accuracies of the two models

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, the dropout is performing regularization. As we can see that the validation accuracy for two drops are similar while the train accuracy for dropout = 1 is higher, the model without dropout is overfitting the samples. Therefore, we can tell that the dropout is forcing the model to be more generalised and is performing 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 [16]:

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



solvers = {}

model = FullyConnectedNet([500, 500, 500, 500], weight_scale=0.03,  dropout=0.8, use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=25, batch_size=200,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay = 0.95,
                verbose=True, print_every=100)
solver.train()


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

```
(Iteration 1 / 6125) loss: 2.378138
(Epoch 0 / 25) train acc: 0.135000; val_acc: 0.150000
(Iteration 101 / 6125) loss: 1.605421
(Iteration 201 / 6125) loss: 1.542825
(Epoch 1 / 25) train acc: 0.476000; val_acc: 0.474000
(Iteration 301 / 6125) loss: 1.538562
(Iteration 401 / 6125) loss: 1.443305
(Epoch 2 / 25) train acc: 0.524000; val_acc: 0.490000
(Iteration 501 / 6125) loss: 1.398444
(Iteration 601 / 6125) loss: 1.372567
(Iteration 701 / 6125) loss: 1.280711
(Epoch 3 / 25) train acc: 0.553000; val_acc: 0.519000
(Iteration 801 / 6125) loss: 1.318911
(Iteration 901 / 6125) loss: 1.319420
(Epoch 4 / 25) train acc: 0.555000; val_acc: 0.553000
(Iteration 1001 / 6125) loss: 1.352377
(Iteration 1101 / 6125) loss: 1.217266
(Iteration 1201 / 6125) loss: 1.322373
(Epoch 5 / 25) train acc: 0.618000; val_acc: 0.549000
(Iteration 1301 / 6125) loss: 1.207786
(Iteration 1401 / 6125) loss: 1.103696
(Epoch 6 / 25) train acc: 0.620000; val_acc: 0.550000
(Iteration 1501 / 6125) loss: 1.177861
(Iteration 1601 / 6125) loss: 1.125841
(Iteration 1701 / 6125) loss: 1.151310
(Epoch 7 / 25) train acc: 0.662000; val_acc: 0.562000
(Iteration 1801 / 6125) loss: 1.199407
(Iteration 1901 / 6125) loss: 1.039357
(Epoch 8 / 25) train acc: 0.697000; val_acc: 0.559000
(Iteration 2001 / 6125) loss: 1.100534
(Iteration 2101 / 6125) loss: 1.104138
(Iteration 2201 / 6125) loss: 1.008565
(Epoch 9 / 25) train acc: 0.692000; val_acc: 0.581000
(Iteration 2301 / 6125) loss: 1.074277
(Iteration 2401 / 6125) loss: 1.062424
(Epoch 10 / 25) train acc: 0.689000; val_acc: 0.582000
(Iteration 2501 / 6125) loss: 0.999744
(Iteration 2601 / 6125) loss: 0.920426
(Epoch 11 / 25) train acc: 0.738000; val_acc: 0.589000
(Iteration 2701 / 6125) loss: 0.969963
(Iteration 2801 / 6125) loss: 0.858363
(Iteration 2901 / 6125) loss: 0.969156
(Epoch 12 / 25) train acc: 0.738000; val_acc: 0.594000
(Iteration 3001 / 6125) loss: 0.916767
(Iteration 3101 / 6125) loss: 0.992136
(Epoch 13 / 25) train acc: 0.746000; val_acc: 0.586000
(Iteration 3201 / 6125) loss: 0.809947
(Iteration 3301 / 6125) loss: 0.806089
(Iteration 3401 / 6125) loss: 0.800861
(Epoch 14 / 25) train acc: 0.757000; val_acc: 0.588000
(Iteration 3501 / 6125) loss: 0.676210
(Iteration 3601 / 6125) loss: 0.942014
(Epoch 15 / 25) train acc: 0.784000; val_acc: 0.600000
(Iteration 3701 / 6125) loss: 0.830425
(Iteration 3801 / 6125) loss: 0.839415
(Iteration 3901 / 6125) loss: 0.828776
(Epoch 16 / 25) train acc: 0.803000; val_acc: 0.602000
(Iteration 4001 / 6125) loss: 0.744202
(Iteration 4101 / 6125) loss: 0.800064
(Epoch 17 / 25) train acc: 0.820000; val_acc: 0.592000
(Iteration 4201 / 6125) loss: 0.868332
(Iteration 4301 / 6125) loss: 0.801582
(Iteration 4401 / 6125) loss: 0.774704
(Epoch 18 / 25) train acc: 0.812000; val_acc: 0.589000
(Iteration 4501 / 6125) loss: 0.717396
(Iteration 4601 / 6125) loss: 0.869917
(Epoch 19 / 25) train acc: 0.796000; val_acc: 0.609000
(Iteration 4701 / 6125) loss: 0.690982
(Iteration 4801 / 6125) loss: 0.699279
(Epoch 20 / 25) train acc: 0.844000; val_acc: 0.586000
(Iteration 4901 / 6125) loss: 0.714194
(Iteration 5001 / 6125) loss: 0.864611
(Iteration 5101 / 6125) loss: 0.790376
(Epoch 21 / 25) train acc: 0.836000; val_acc: 0.604000
(Iteration 5201 / 6125) loss: 0.698636
(Iteration 5301 / 6125) loss: 0.612872
(Epoch 22 / 25) train acc: 0.850000; val_acc: 0.592000
(Iteration 5401 / 6125) loss: 0.705771
(Iteration 5501 / 6125) loss: 0.745434
(Iteration 5601 / 6125) loss: 0.653437
(Epoch 23 / 25) train acc: 0.855000; val_acc: 0.599000
(Iteration 5701 / 6125) loss: 0.513506
(Iteration 5801 / 6125) loss: 0.552587
(Epoch 24 / 25) train acc: 0.880000; val_acc: 0.604000
(Iteration 5901 / 6125) loss: 0.585078
(Iteration 6001 / 6125) loss: 0.539350
(Iteration 6101 / 6125) loss: 0.680519
(Epoch 25 / 25) train acc: 0.879000; val_acc: 0.593000
```