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 [ ]:
```
## 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))))
```

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

In [ ]:
```
# 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 [ ]:
```
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:  10.001199068997368
Mean of train-time output:  10.0700616634419
Mean of test-time output:  10.001199068997368
Fraction of train-time output set to zero:  0.697984
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.001199068997368
Mean of train-time output:  10.017545453767633
Mean of test-time output:  10.001199068997368
Fraction of train-time output set to zero:  0.398944
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.001199068997368
Mean of train-time output:  9.999852007144597
Mean of test-time output:  10.001199068997368
Fraction of train-time output set to zero:  0.250056
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 [ ]:
```
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:  5.445610892205213e-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 [ ]:
```
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.

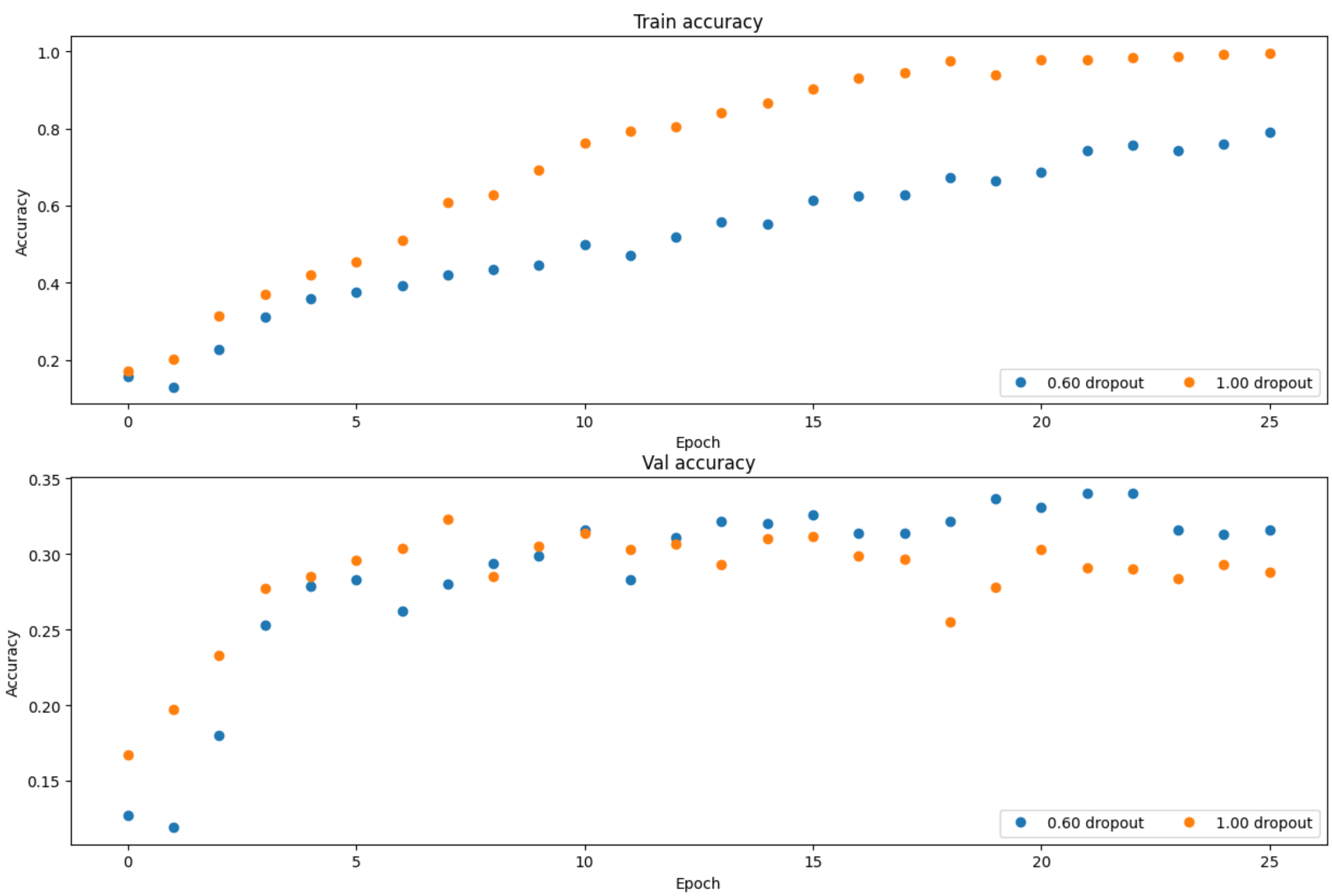
In [ ]:
```
# Train two identical nets, one with dropout and one without

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: 0.130000; val_acc: 0.119000
(Epoch 2 / 25) train acc: 0.226000; val_acc: 0.180000
(Epoch 3 / 25) train acc: 0.312000; val_acc: 0.253000
(Epoch 4 / 25) train acc: 0.360000; val_acc: 0.279000
(Epoch 5 / 25) train acc: 0.376000; val_acc: 0.283000
(Epoch 6 / 25) train acc: 0.392000; val_acc: 0.262000
(Epoch 7 / 25) train acc: 0.422000; val_acc: 0.280000
(Epoch 8 / 25) train acc: 0.436000; val_acc: 0.294000
(Epoch 9 / 25) train acc: 0.446000; val_acc: 0.299000
(Epoch 10 / 25) train acc: 0.498000; val_acc: 0.316000
(Epoch 11 / 25) train acc: 0.470000; val_acc: 0.283000
(Epoch 12 / 25) train acc: 0.520000; val_acc: 0.311000
(Epoch 13 / 25) train acc: 0.558000; val_acc: 0.322000
(Epoch 14 / 25) train acc: 0.552000; val_acc: 0.320000
(Epoch 15 / 25) train acc: 0.614000; val_acc: 0.326000
(Epoch 16 / 25) train acc: 0.626000; val_acc: 0.314000
(Epoch 17 / 25) train acc: 0.628000; val_acc: 0.314000
(Epoch 18 / 25) train acc: 0.674000; val_acc: 0.322000
(Epoch 19 / 25) train acc: 0.664000; val_acc: 0.337000
(Epoch 20 / 25) train acc: 0.686000; val_acc: 0.331000
(Iteration 101 / 125) loss: 1.211559
(Epoch 21 / 25) train acc: 0.744000; val_acc: 0.340000
(Epoch 22 / 25) train acc: 0.758000; val_acc: 0.340000
(Epoch 23 / 25) train acc: 0.742000; val_acc: 0.316000
(Epoch 24 / 25) train acc: 0.760000; val_acc: 0.313000
(Epoch 25 / 25) train acc: 0.790000; val_acc: 0.316000
(Iteration 1 / 125) loss: 2.300607
(Epoch 0 / 25) train acc: 0.172000; val_acc: 0.167000
(Epoch 1 / 25) train acc: 0.202000; val_acc: 0.197000
(Epoch 2 / 25) train acc: 0.314000; val_acc: 0.233000
(Epoch 3 / 25) train acc: 0.370000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.420000; val_acc: 0.285000
(Epoch 5 / 25) train acc: 0.454000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.510000; val_acc: 0.304000
(Epoch 7 / 25) train acc: 0.608000; val_acc: 0.323000
(Epoch 8 / 25) train acc: 0.628000; val_acc: 0.285000
(Epoch 9 / 25) train acc: 0.694000; val_acc: 0.305000
(Epoch 10 / 25) train acc: 0.764000; val_acc: 0.314000
(Epoch 11 / 25) train acc: 0.794000; val_acc: 0.303000
(Epoch 12 / 25) train acc: 0.804000; val_acc: 0.307000
(Epoch 13 / 25) train acc: 0.840000; val_acc: 0.293000
(Epoch 14 / 25) train acc: 0.866000; val_acc: 0.310000
(Epoch 15 / 25) train acc: 0.904000; val_acc: 0.312000
(Epoch 16 / 25) train acc: 0.930000; val_acc: 0.299000
(Epoch 17 / 25) train acc: 0.944000; val_acc: 0.297000
(Epoch 18 / 25) train acc: 0.976000; val_acc: 0.255000
(Epoch 19 / 25) train acc: 0.940000; val_acc: 0.278000
(Epoch 20 / 25) train acc: 0.980000; val_acc: 0.303000
(Iteration 101 / 125) loss: 0.087768
(Epoch 21 / 25) train acc: 0.980000; val_acc: 0.291000
(Epoch 22 / 25) train acc: 0.984000; val_acc: 0.290000
(Epoch 23 / 25) train acc: 0.986000; val_acc: 0.284000
(Epoch 24 / 25) train acc: 0.992000; val_acc: 0.293000
(Epoch 25 / 25) train acc: 0.996000; val_acc: 0.288000
```

In [ ]:

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

train_accs = []
val_accs = []
for dropout in dropout_choices:
```

```python
    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:

The orange when dropout = 1 represents not using dropout. So when dropout = 0.6, it is being used. So the blue represents dropout. Based on the experiment, yes dropout performs regularization because you can see both blue and orange have similar validation accuracy while the blue (dropout) doesn't have such a high rate of increase for train accuracy. This means that it is mitigating overfitting and 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 [ ]:

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

optimizer = 'adam'
best_model = None

layer_dims = [500, 500, 500]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale,
                          use_batchnorm=True, dropout=0.6)

solver = Solver(model, data,
                num_epochs=10, batch_size=100,
                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 / 4900) loss: 2.315670
(Epoch 0 / 10) train acc: 0.116000; val_acc: 0.116000
(Iteration 51 / 4900) loss: 1.942495
(Iteration 101 / 4900) loss: 1.732891
(Iteration 151 / 4900) loss: 1.610360
(Iteration 201 / 4900) loss: 1.794659
(Iteration 251 / 4900) loss: 1.614585
(Iteration 301 / 4900) loss: 1.614509
(Iteration 351 / 4900) loss: 1.440573
(Iteration 401 / 4900) loss: 1.639682
(Iteration 451 / 4900) loss: 1.613830
(Epoch 1 / 10) train acc: 0.452000; val_acc: 0.477000
(Iteration 501 / 4900) loss: 1.388291
(Iteration 551 / 4900) loss: 1.400066
(Iteration 601 / 4900) loss: 1.555558
(Iteration 651 / 4900) loss: 1.682169
(Iteration 701 / 4900) loss: 1.325679
(Iteration 751 / 4900) loss: 1.399323
(Iteration 801 / 4900) loss: 1.459228
(Iteration 851 / 4900) loss: 1.584442
(Iteration 901 / 4900) loss: 1.408745
(Iteration 951 / 4900) loss: 1.515232
(Epoch 2 / 10) train acc: 0.502000; val_acc: 0.488000
(Iteration 1001 / 4900) loss: 1.653320
(Iteration 1051 / 4900) loss: 1.454258
(Iteration 1101 / 4900) loss: 1.556873
(Iteration 1151 / 4900) loss: 1.539827
(Iteration 1201 / 4900) loss: 1.564294
(Iteration 1251 / 4900) loss: 1.226554
(Iteration 1301 / 4900) loss: 1.402670
(Iteration 1351 / 4900) loss: 1.310938
(Iteration 1401 / 4900) loss: 1.512348
(Iteration 1451 / 4900) loss: 1.405581
(Epoch 3 / 10) train acc: 0.553000; val_acc: 0.508000
(Iteration 1501 / 4900) loss: 1.431631
(Iteration 1551 / 4900) loss: 1.751771
(Iteration 1601 / 4900) loss: 1.314030
(Iteration 1651 / 4900) loss: 1.460319
(Iteration 1701 / 4900) loss: 1.384233
(Iteration 1751 / 4900) loss: 1.322205
(Iteration 1801 / 4900) loss: 1.355316
(Iteration 1851 / 4900) loss: 1.460610
(Iteration 1901 / 4900) loss: 1.445547
(Iteration 1951 / 4900) loss: 1.243986
(Epoch 4 / 10) train acc: 0.557000; val_acc: 0.537000
(Iteration 2001 / 4900) loss: 1.329682
(Iteration 2051 / 4900) loss: 1.259213
(Iteration 2101 / 4900) loss: 1.332589
(Iteration 2151 / 4900) loss: 1.332359
(Iteration 2201 / 4900) loss: 1.337638
(Iteration 2251 / 4900) loss: 1.124938
(Iteration 2301 / 4900) loss: 1.390980
(Iteration 2351 / 4900) loss: 1.345582
(Iteration 2401 / 4900) loss: 1.219983
(Epoch 5 / 10) train acc: 0.585000; val_acc: 0.546000
(Iteration 2451 / 4900) loss: 1.234222
(Iteration 2501 / 4900) loss: 1.251678
(Iteration 2551 / 4900) loss: 1.223066
(Iteration 2601 / 4900) loss: 1.093901
(Iteration 2651 / 4900) loss: 1.327221
(Iteration 2701 / 4900) loss: 1.168672
(Iteration 2751 / 4900) loss: 1.412426
```

```
(Iteration 2801 / 4900) loss: 1.127916
(Iteration 2851 / 4900) loss: 1.145958
(Iteration 2901 / 4900) loss: 1.297254
(Epoch 6 / 10) train acc: 0.597000; val_acc: 0.563000
(Iteration 2951 / 4900) loss: 1.348911
(Iteration 3001 / 4900) loss: 1.135094
(Iteration 3051 / 4900) loss: 1.212895
(Iteration 3101 / 4900) loss: 1.289152
(Iteration 3151 / 4900) loss: 1.243169
(Iteration 3201 / 4900) loss: 1.125017
(Iteration 3251 / 4900) loss: 1.283074
(Iteration 3301 / 4900) loss: 1.278779
(Iteration 3351 / 4900) loss: 1.190933
(Iteration 3401 / 4900) loss: 1.438955
(Epoch 7 / 10) train acc: 0.592000; val_acc: 0.555000
(Iteration 3451 / 4900) loss: 1.300583
(Iteration 3501 / 4900) loss: 1.444260
(Iteration 3551 / 4900) loss: 1.221397
(Iteration 3601 / 4900) loss: 1.090857
(Iteration 3651 / 4900) loss: 1.240972
(Iteration 3701 / 4900) loss: 1.373810
(Iteration 3751 / 4900) loss: 1.215900
(Iteration 3801 / 4900) loss: 1.163014
(Iteration 3851 / 4900) loss: 1.281768
(Iteration 3901 / 4900) loss: 1.124513
(Epoch 8 / 10) train acc: 0.629000; val_acc: 0.564000
(Iteration 3951 / 4900) loss: 1.246855
(Iteration 4001 / 4900) loss: 1.117939
(Iteration 4051 / 4900) loss: 1.240018
(Iteration 4101 / 4900) loss: 1.378874
(Iteration 4151 / 4900) loss: 1.297547
(Iteration 4201 / 4900) loss: 1.206859
(Iteration 4251 / 4900) loss: 1.295331
(Iteration 4301 / 4900) loss: 1.269939
(Iteration 4351 / 4900) loss: 1.419047
(Iteration 4401 / 4900) loss: 1.023463
(Epoch 9 / 10) train acc: 0.610000; val_acc: 0.564000
(Iteration 4451 / 4900) loss: 1.261891
(Iteration 4501 / 4900) loss: 1.186561
(Iteration 4551 / 4900) loss: 1.296652
(Iteration 4601 / 4900) loss: 1.202455
(Iteration 4651 / 4900) loss: 1.140189
(Iteration 4701 / 4900) loss: 1.165684
(Iteration 4751 / 4900) loss: 1.264138
(Iteration 4801 / 4900) loss: 1.149772
(Iteration 4851 / 4900) loss: 1.164538
(Epoch 10 / 10) train acc: 0.652000; val_acc: 0.564000
```

Validation accuracy reached 56.4%!!

In [ ]: