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 utils.data_utils import get_CIFAR10_data
         from utils.gradient_check import eval_numerical_gradient, eval_numerical_gradient_a
         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 [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).mean())
           print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
```

```
Running tests with p =  0.3
Mean of input:  10.003225190648825
Mean of train-time output:  10.011147338919587
Mean of test-time output:  10.003225190648825
Fraction of train-time output set to zero:  0.699776
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.003225190648825
Mean of train-time output:  9.978352959114723
Mean of test-time output:  10.003225190648825
Fraction of train-time output set to zero:  0.40156
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.003225190648825
Mean of train-time output:  10.000808346367215
Mean of test-time output:  10.003225190648825
Fraction of train-time output set to zero:  0.250124
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 [5]:  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

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

```
dx relative error:  5.445610771836907e-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 [13]:
```python
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: 4.795196815215288e-07
W2 relative error: 1.9717710574314515e-07
W3 relative error: 1.5587099483501822e-07
b1 relative error: 2.033615448560775e-08
b2 relative error: 1.686315567518667e-09
b3 relative error: 1.1144421861081857e-10


Running check with dropout =  0.25
Initial loss:   2.3161871104971326
W1 relative error: 3.2340141421343043e-07
W2 relative error: 4.813124944464727e-09
W3 relative error: 2.5076237000747585e-09
b1 relative error: 3.817580030628832e-08
b2 relative error: 1.277703658392353e-10
b3 relative error: 6.263703890666123e-11


Running check with dropout =  0.5
Initial loss:   2.3040458699680304
W1 relative error: 6.150172004305114e-07
W2 relative error: 4.353961611622114e-08
W3 relative error: 9.77483271160455e-08
b1 relative error: 3.966755772218227e-08
b2 relative error: 6.245710242676597e-10
b3 relative error: 1.2392033987263167e-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.

```python
In [24]:  # 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',
```

```python
                optim_config={
                  'learning_rate': 5e-4,
                },
                verbose=True, print_every=100)
solver.train()
solvers[dropout] = solver
```

```
(Iteration 1 / 125) loss: 2.300903
(Epoch 0 / 25) train acc: 0.186000; val_acc: 0.159000
(Epoch 1 / 25) train acc: 0.228000; val_acc: 0.190000
(Epoch 2 / 25) train acc: 0.258000; val_acc: 0.201000
(Epoch 3 / 25) train acc: 0.310000; val_acc: 0.244000
(Epoch 4 / 25) train acc: 0.350000; val_acc: 0.263000
(Epoch 5 / 25) train acc: 0.370000; val_acc: 0.279000
(Epoch 6 / 25) train acc: 0.440000; val_acc: 0.306000
(Epoch 7 / 25) train acc: 0.480000; val_acc: 0.307000
(Epoch 8 / 25) train acc: 0.508000; val_acc: 0.311000
(Epoch 9 / 25) train acc: 0.596000; val_acc: 0.311000
(Epoch 10 / 25) train acc: 0.608000; val_acc: 0.332000
(Epoch 11 / 25) train acc: 0.690000; val_acc: 0.304000
(Epoch 12 / 25) train acc: 0.736000; val_acc: 0.288000
(Epoch 13 / 25) train acc: 0.780000; val_acc: 0.307000
(Epoch 14 / 25) train acc: 0.812000; val_acc: 0.301000
(Epoch 15 / 25) train acc: 0.814000; val_acc: 0.291000
(Epoch 16 / 25) train acc: 0.862000; val_acc: 0.299000
(Epoch 17 / 25) train acc: 0.898000; val_acc: 0.277000
(Epoch 18 / 25) train acc: 0.932000; val_acc: 0.297000
(Epoch 19 / 25) train acc: 0.946000; val_acc: 0.273000
(Epoch 20 / 25) train acc: 0.966000; val_acc: 0.317000
(Iteration 101 / 125) loss: 0.129813
(Epoch 21 / 25) train acc: 0.978000; val_acc: 0.306000
(Epoch 22 / 25) train acc: 0.986000; val_acc: 0.273000
(Epoch 23 / 25) train acc: 0.992000; val_acc: 0.298000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.303000
(Epoch 25 / 25) train acc: 0.996000; val_acc: 0.316000
(Iteration 1 / 125) loss: 2.301961
(Epoch 0 / 25) train acc: 0.178000; val_acc: 0.128000
(Epoch 1 / 25) train acc: 0.154000; val_acc: 0.146000
(Epoch 2 / 25) train acc: 0.248000; val_acc: 0.203000
(Epoch 3 / 25) train acc: 0.300000; val_acc: 0.224000
(Epoch 4 / 25) train acc: 0.326000; val_acc: 0.237000
(Epoch 5 / 25) train acc: 0.376000; val_acc: 0.275000
(Epoch 6 / 25) train acc: 0.416000; val_acc: 0.292000
(Epoch 7 / 25) train acc: 0.438000; val_acc: 0.301000
(Epoch 8 / 25) train acc: 0.520000; val_acc: 0.308000
(Epoch 9 / 25) train acc: 0.544000; val_acc: 0.319000
(Epoch 10 / 25) train acc: 0.574000; val_acc: 0.330000
(Epoch 11 / 25) train acc: 0.596000; val_acc: 0.322000
(Epoch 12 / 25) train acc: 0.636000; val_acc: 0.331000
(Epoch 13 / 25) train acc: 0.700000; val_acc: 0.340000
(Epoch 14 / 25) train acc: 0.718000; val_acc: 0.331000
(Epoch 15 / 25) train acc: 0.736000; val_acc: 0.306000
(Epoch 16 / 25) train acc: 0.786000; val_acc: 0.330000
(Epoch 17 / 25) train acc: 0.820000; val_acc: 0.319000
(Epoch 18 / 25) train acc: 0.852000; val_acc: 0.324000
(Epoch 19 / 25) train acc: 0.860000; val_acc: 0.314000
(Epoch 20 / 25) train acc: 0.872000; val_acc: 0.319000
(Iteration 101 / 125) loss: 0.421533
(Epoch 21 / 25) train acc: 0.912000; val_acc: 0.331000
(Epoch 22 / 25) train acc: 0.928000; val_acc: 0.301000
(Epoch 23 / 25) train acc: 0.960000; val_acc: 0.299000
(Epoch 24 / 25) train acc: 0.972000; val_acc: 0.298000
(Epoch 25 / 25) train acc: 0.960000; val_acc: 0.313000
```

```
In [25]:  # 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()
```
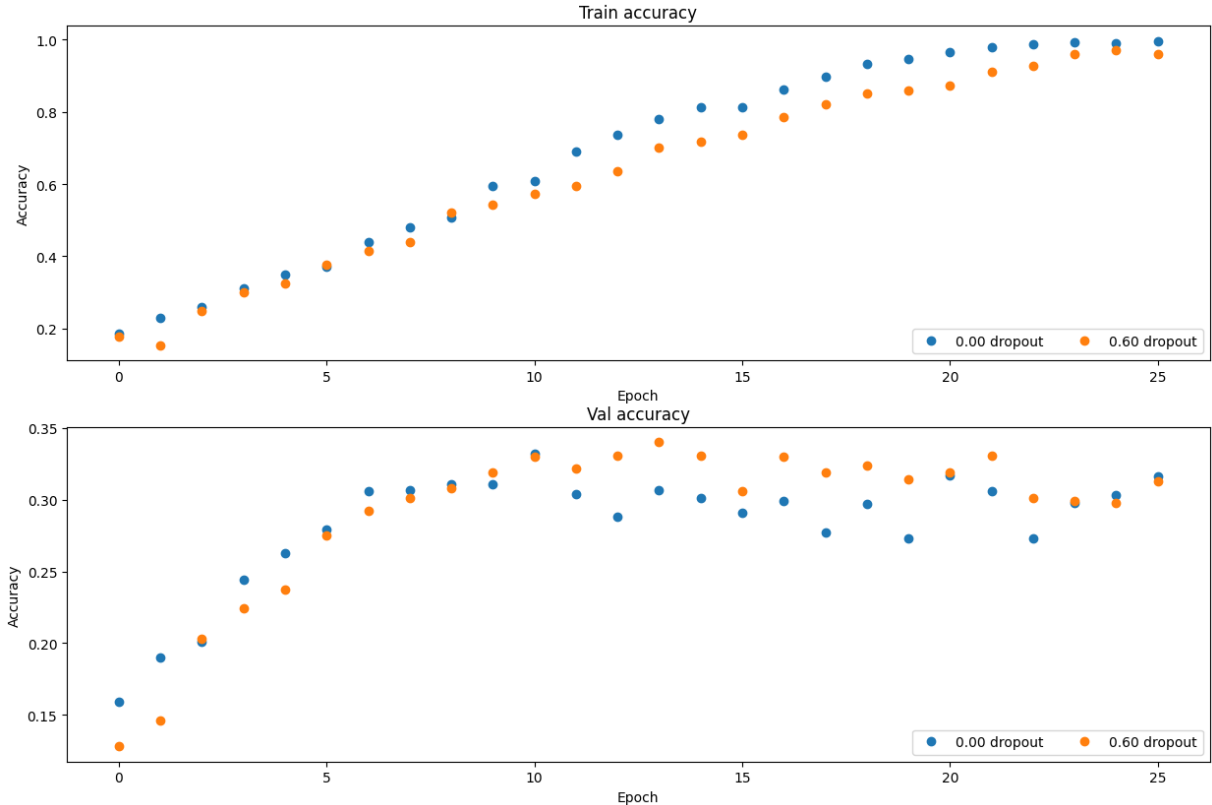

Question

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

Answer:

Dropout is performing regularization. We can tell from the graphs that the model using dropout has a lower training accuracy than the model without, but has a generally higher validation accuracy. This is because regularizing the model allows it to generalize better.

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 [76]:  # ================================================================ #
          # YOUR CODE HERE:
          #    Implement a FC-net that achieves at least 55% validation accuracy
          #    on CIFAR-10.
          # ================================================================ #
          optimizer = 'adam'
          best_model = None

          best_dim = [600,600,600]
          weight_scale = 0.005
          best_lr   = 0.0001
          best_dropout = 0.7
          best_decay = 0.85

          model = FullyConnectedNet(best_dim, weight_scale=weight_scale,
                                use_batchnorm=True ,dropout=best_dropout)

          solver = Solver(model, data,
                      num_epochs=10, batch_size=100,
                      update_rule=optimizer,
                      optim_config={
                      'learning_rate': best_lr,
                      },
                      lr_decay=best_decay,
                      verbose=True, print_every = 10000)
          solver.train()
          # ================================================================ #
          # END YOUR CODE HERE
          # ================================================================ #
```

```
(Iteration 1 / 4900) loss: 2.327201
(Epoch 0 / 10) train acc: 0.168000; val_acc: 0.181000
(Epoch 1 / 10) train acc: 0.383000; val_acc: 0.427000
(Epoch 2 / 10) train acc: 0.465000; val_acc: 0.441000
(Epoch 3 / 10) train acc: 0.475000; val_acc: 0.473000
(Epoch 4 / 10) train acc: 0.492000; val_acc: 0.488000
(Epoch 5 / 10) train acc: 0.551000; val_acc: 0.494000
(Epoch 6 / 10) train acc: 0.569000; val_acc: 0.516000
(Epoch 7 / 10) train acc: 0.580000; val_acc: 0.521000
(Epoch 8 / 10) train acc: 0.569000; val_acc: 0.518000
(Epoch 9 / 10) train acc: 0.646000; val_acc: 0.513000
(Epoch 10 / 10) train acc: 0.639000; val_acc: 0.543000
```