Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some output activations to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

Acknowledgement: This exercise is adapted from Stanford CS231n.

In [1]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.fc_net import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from libs.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, v in data.items():
  print('%s: ' % k, v.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

In the file `libs/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

In [4]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  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())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `libs/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

In [5]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, '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)

# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

Fully-connected nets with Dropout

In the file `libs/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the network receives a value that is not 1 for the `dropout` parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

In [6]:
```
np.random.seed(231)
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 [1, 0.75, 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)

  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  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('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  0.0

Running check with dropout =  0.75
Initial loss:  0.0

Running check with dropout =  0.5
Initial loss:  0.0
```

Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

In [9]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
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 = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='sgd',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
  print()
```

```
1
(Iteration 1 / 125) loss: 7.856644
(Epoch 0 / 25) train acc: 0.166000; val_acc: 0.143000
(Epoch 1 / 25) train acc: 0.226000; val_acc: 0.183000
(Epoch 2 / 25) train acc: 0.380000; val_acc: 0.220000
(Epoch 3 / 25) train acc: 0.458000; val_acc: 0.253000
(Epoch 4 / 25) train acc: 0.584000; val_acc: 0.259000
(Epoch 5 / 25) train acc: 0.638000; val_acc: 0.264000
(Epoch 6 / 25) train acc: 0.648000; val_acc: 0.268000
(Epoch 7 / 25) train acc: 0.766000; val_acc: 0.269000
(Epoch 8 / 25) train acc: 0.780000; val_acc: 0.281000
(Epoch 9 / 25) train acc: 0.740000; val_acc: 0.247000
(Epoch 10 / 25) train acc: 0.932000; val_acc: 0.282000
(Epoch 11 / 25) train acc: 0.966000; val_acc: 0.278000
(Epoch 12 / 25) train acc: 0.984000; val_acc: 0.277000
(Epoch 13 / 25) train acc: 0.988000; val_acc: 0.277000
(Epoch 14 / 25) train acc: 0.994000; val_acc: 0.277000
(Epoch 15 / 25) train acc: 0.998000; val_acc: 0.287000
(Epoch 16 / 25) train acc: 0.998000; val_acc: 0.279000
(Epoch 17 / 25) train acc: 0.998000; val_acc: 0.284000
(Epoch 18 / 25) train acc: 0.998000; val_acc: 0.275000
(Epoch 19 / 25) train acc: 1.000000; val_acc: 0.282000
(Epoch 20 / 25) train acc: 1.000000; val_acc: 0.280000
(Iteration 101 / 125) loss: 0.047756
(Epoch 21 / 25) train acc: 1.000000; val_acc: 0.285000
(Epoch 22 / 25) train acc: 1.000000; val_acc: 0.278000
(Epoch 23 / 25) train acc: 1.000000; val_acc: 0.283000
(Epoch 24 / 25) train acc: 1.000000; val_acc: 0.283000
(Epoch 25 / 25) train acc: 1.000000; val_acc: 0.284000

0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.204000; val_acc: 0.169000
(Epoch 1 / 25) train acc: 0.282000; val_acc: 0.200000
(Epoch 2 / 25) train acc: 0.384000; val_acc: 0.225000
(Epoch 3 / 25) train acc: 0.440000; val_acc: 0.246000
(Epoch 4 / 25) train acc: 0.570000; val_acc: 0.275000
(Epoch 5 / 25) train acc: 0.544000; val_acc: 0.277000
(Epoch 6 / 25) train acc: 0.568000; val_acc: 0.255000
(Epoch 7 / 25) train acc: 0.668000; val_acc: 0.273000
(Epoch 8 / 25) train acc: 0.678000; val_acc: 0.276000
(Epoch 9 / 25) train acc: 0.716000; val_acc: 0.290000
(Epoch 10 / 25) train acc: 0.732000; val_acc: 0.312000
(Epoch 11 / 25) train acc: 0.760000; val_acc: 0.291000
(Epoch 12 / 25) train acc: 0.750000; val_acc: 0.293000
(Epoch 13 / 25) train acc: 0.766000; val_acc: 0.284000
(Epoch 14 / 25) train acc: 0.794000; val_acc: 0.308000
(Epoch 15 / 25) train acc: 0.824000; val_acc: 0.323000
(Epoch 16 / 25) train acc: 0.800000; val_acc: 0.298000
(Epoch 17 / 25) train acc: 0.824000; val_acc: 0.315000
(Epoch 18 / 25) train acc: 0.826000; val_acc: 0.319000
(Epoch 19 / 25) train acc: 0.852000; val_acc: 0.322000
(Epoch 20 / 25) train acc: 0.832000; val_acc: 0.277000
(Iteration 101 / 125) loss: 2.363841
(Epoch 21 / 25) train acc: 0.840000; val_acc: 0.301000
(Epoch 22 / 25) train acc: 0.890000; val_acc: 0.316000
(Epoch 23 / 25) train acc: 0.870000; val_acc: 0.302000
(Epoch 24 / 25) train acc: 0.914000; val_acc: 0.296000
(Epoch 25 / 25) train acc: 0.892000; val_acc: 0.290000
```

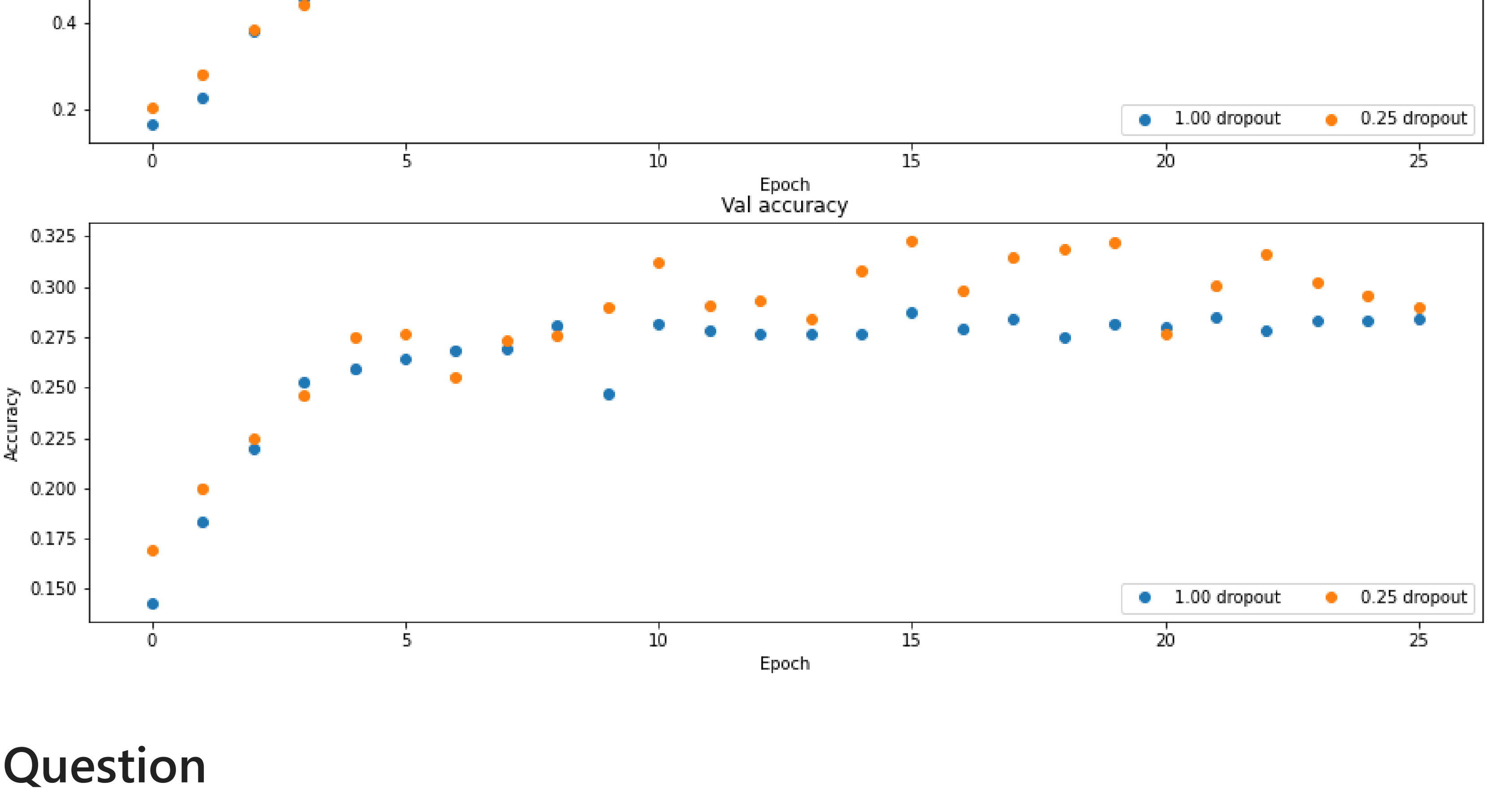
In [10]:
```
# Plot train and validation accuracies of the two models

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

Explain what you see in this experiment. What does it suggest about **dropout**?

In this experiment, we can see that both models are overfitting, where both training accuracies are much higher than the validation accuracies.

Without dropout:
There is a huge difference between training accuracy (100% at epoch 25) and validation accuracy (28.4% at epoch 25), hence indicating that the model is overfitting.

With dropout:
The training accuracy decreases (89.2% at epoch 25) but validation accuracy increases (29% at epoch 25)

Conclusion:
This shows that dropout helps to regularize the model to attempt to reduce overfitting. As some nodes are randomly dropped during the training process, this forces nodes within a layer to probabilistically take on varying responsibility for the inputs. This encourages the model to learn sparse representations which can help to reduce overfitting.