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 (https://arxiv.org/abs/1207.0580)

```
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 cs231n.classifiers.fc_net import *
         from cs231n.data_utils import get_CIFAR10_data
         from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradi
         ent_array
         from cs231n.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-ipyth
         on
         %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))))
```

run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel

```
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 `cs231n/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 [3]:  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:  9.99559079897757
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.250216
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:  10.01506802495506
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.399204
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:  10.029131799886338
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.69926
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

```
In [4]:  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_p
         aram)[0], x, dout)

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

```
dx relative error:  5.445612718272284e-11
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

Answer:

If we do not divide the values by p then while testing we won't consider the average training output. Moreover, we will only consider the sum of all possible sub-networks which may lead to high values. This will occur because at testing we require approximate expected outputs produced by the training phase, because of this we will only perform forward calls without dropping neurons out.

Fully-connected nets with Dropout

In the file `cs231n/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 [5]:  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:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 1.69e-10

Running check with dropout =  0.75
Initial loss:  2.3001748924793235
W1 relative error: 3.05e-08
W2 relative error: 2.05e-09
W3 relative error: 1.93e-09
b1 relative error: 8.86e-10
b2 relative error: 3.33e-01
b3 relative error: 8.54e-11

Running check with dropout =  0.5
Initial loss:  2.310136908722148
W1 relative error: 2.57e-08
W2 relative error: 1.49e-08
W3 relative error: 4.49e-08
b1 relative error: 3.93e-10
b2 relative error: 1.91e-09
b3 relative error: 9.51e-11
```

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 [6]:  # 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='adam',
                           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: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.269000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.275000
(Epoch 10 / 25) train acc: 0.890000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.930000; val_acc: 0.283000
(Epoch 12 / 25) train acc: 0.958000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.964000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.962000; val_acc: 0.317000
(Epoch 15 / 25) train acc: 0.962000; val_acc: 0.304000
(Epoch 16 / 25) train acc: 0.980000; val_acc: 0.307000
(Epoch 17 / 25) train acc: 0.970000; val_acc: 0.321000
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.317000
(Epoch 19 / 25) train acc: 0.984000; val_acc: 0.303000
(Epoch 20 / 25) train acc: 0.986000; val_acc: 0.309000
(Iteration 101 / 125) loss: 0.074303
(Epoch 21 / 25) train acc: 0.996000; val_acc: 0.304000
(Epoch 22 / 25) train acc: 0.964000; val_acc: 0.311000
(Epoch 23 / 25) train acc: 0.982000; val_acc: 0.314000
(Epoch 24 / 25) train acc: 0.982000; val_acc: 0.305000
(Epoch 25 / 25) train acc: 0.972000; val_acc: 0.305000


0.25
(Iteration 1 / 125) loss: 11.814033
(Epoch 0 / 25) train acc: 0.264000; val_acc: 0.193000
(Epoch 1 / 25) train acc: 0.368000; val_acc: 0.244000
(Epoch 2 / 25) train acc: 0.528000; val_acc: 0.264000
(Epoch 3 / 25) train acc: 0.586000; val_acc: 0.257000
(Epoch 4 / 25) train acc: 0.642000; val_acc: 0.290000
(Epoch 5 / 25) train acc: 0.756000; val_acc: 0.292000
(Epoch 6 / 25) train acc: 0.784000; val_acc: 0.277000
(Epoch 7 / 25) train acc: 0.818000; val_acc: 0.275000
(Epoch 8 / 25) train acc: 0.814000; val_acc: 0.258000
(Epoch 9 / 25) train acc: 0.880000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.908000; val_acc: 0.293000
(Epoch 11 / 25) train acc: 0.918000; val_acc: 0.290000
(Epoch 12 / 25) train acc: 0.922000; val_acc: 0.269000
(Epoch 13 / 25) train acc: 0.950000; val_acc: 0.304000
(Epoch 14 / 25) train acc: 0.924000; val_acc: 0.306000
(Epoch 15 / 25) train acc: 0.928000; val_acc: 0.298000
(Epoch 16 / 25) train acc: 0.938000; val_acc: 0.281000
(Epoch 17 / 25) train acc: 0.972000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.932000; val_acc: 0.297000
(Epoch 19 / 25) train acc: 0.958000; val_acc: 0.295000
(Epoch 20 / 25) train acc: 0.958000; val_acc: 0.299000
(Iteration 101 / 125) loss: 0.398042
(Epoch 21 / 25) train acc: 0.970000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.946000; val_acc: 0.281000
(Epoch 23 / 25) train acc: 0.962000; val_acc: 0.304000
(Epoch 24 / 25) train acc: 0.972000; val_acc: 0.317000
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.288000
```

In [7]:
```python
# 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' % drop
out)
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' % dropou
t)
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()
```
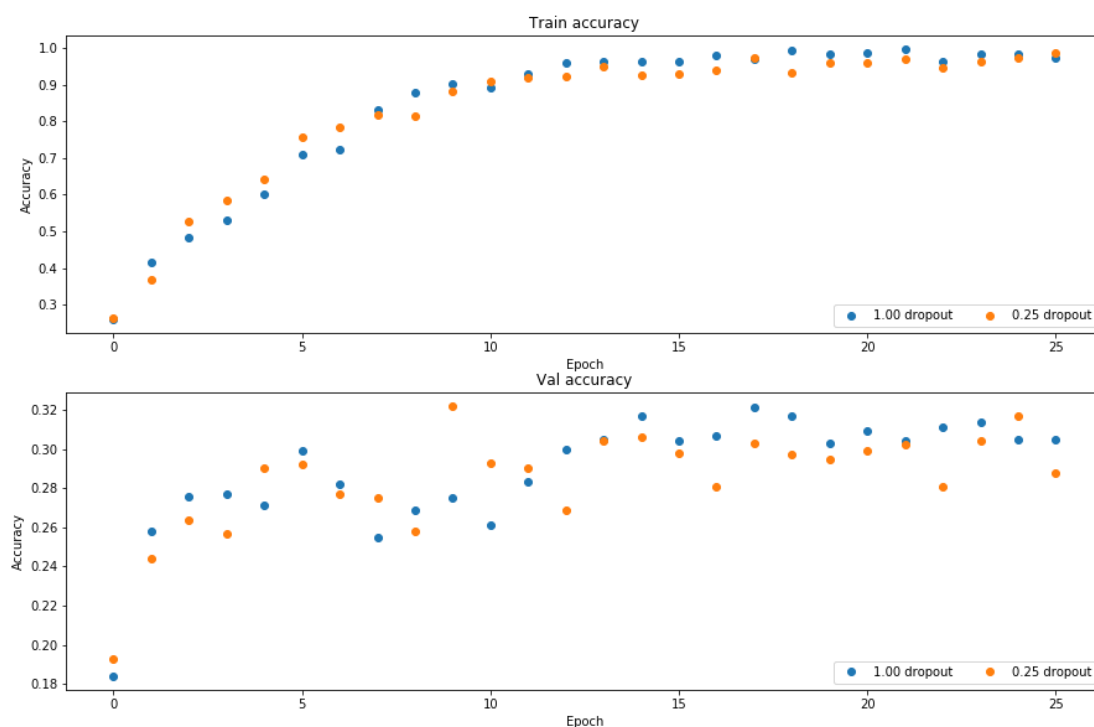
Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

The graph shows us we are overfitting the model. In the training phase, when we do not use dropout, the accuracy is very high however, with dropout the accuracy is smaller. This shows us that with a dropout we are learning a simpler model and as a result trying to avoid overfitting. During our validation, it can be seen that having dropout we managed to get better results. This shows us with dropout we regularize our model and we are reducing the overfit.

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). If we are concerned about overfitting, how should we modify p (if at all) when we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

If we want decrease the amount of the hidden layers, we don't need to modify p because of the number of neurons, which later will be dropped out, will be compared according to the amount of hidden layers. Therefore, we do not need to modify the probability p when we vary the size of the hidden layers.