2 - Dropout

March 11, 2019

1 Dropout

Previously, we have used $l2$-normalization on the network weight for regularizing neural networks. There is another technique to avoid overfitting in training a model.

Dropout [1] is a technique for regularizing neural networks by randomly setting some features 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

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,)
```

2 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 [10]: 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())
             print()
```

```
Running tests with p =  0.3
Mean of input:  10.00162978333737
Mean of train-time output:  12.030818921421957
Mean of test-time output:  10.00162978333737
Fraction of train-time output set to zero:  0.299672
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.6
Mean of input:  10.00162978333737
Mean of train-time output:  3.994468829771193
Mean of test-time output:  10.00162978333737
Fraction of train-time output set to zero:  0.600656
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.75
Mean of input:  10.00162978333737
```

```
Mean of train-time output:  2.510113250000468
Mean of test-time output:  10.00162978333737
Fraction of train-time output set to zero:  0.748932
Fraction of test-time output set to zero:  0.0
```

3 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 [11]: 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:  1.8928983374134574e-11
```

4 Fully-connected nets with Dropout

In the file `libs/classifiers/fc_net.py`, modify your implementation to use dropout. Specificially, if the constructor the the net receives a nonzero value for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

```
In [12]: 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-
            print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
        print
```

```
Running check with dropout =  0
Initial loss:  2.3051948273987857
W1 relative error: 2.53e-07
W2 relative error: 1.50e-05
W3 relative error: 2.75e-07
b1 relative error: 2.94e-06
b2 relative error: 5.05e-08
b3 relative error: 1.17e-10
Running check with dropout =  0.25
Initial loss:  2.303133728006797
W1 relative error: 8.11e-08
W2 relative error: 3.05e-08
W3 relative error: 2.20e-08
b1 relative error: 6.97e-09
b2 relative error: 4.17e-10
b3 relative error: 1.45e-10
Running check with dropout =  0.5
Initial loss:  2.302818168271519
W1 relative error: 1.36e-06
W2 relative error: 3.00e-07
W3 relative error: 4.23e-08
b1 relative error: 3.13e-07
b2 relative error: 2.62e-09
b3 relative error: 6.76e-11
```

5 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 dropout probability of 0.75. We will then visualize the training and validation accuracies of the two networks over time.

In [13]:
```
# 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, 0.25, 0.5, 0.75]
```

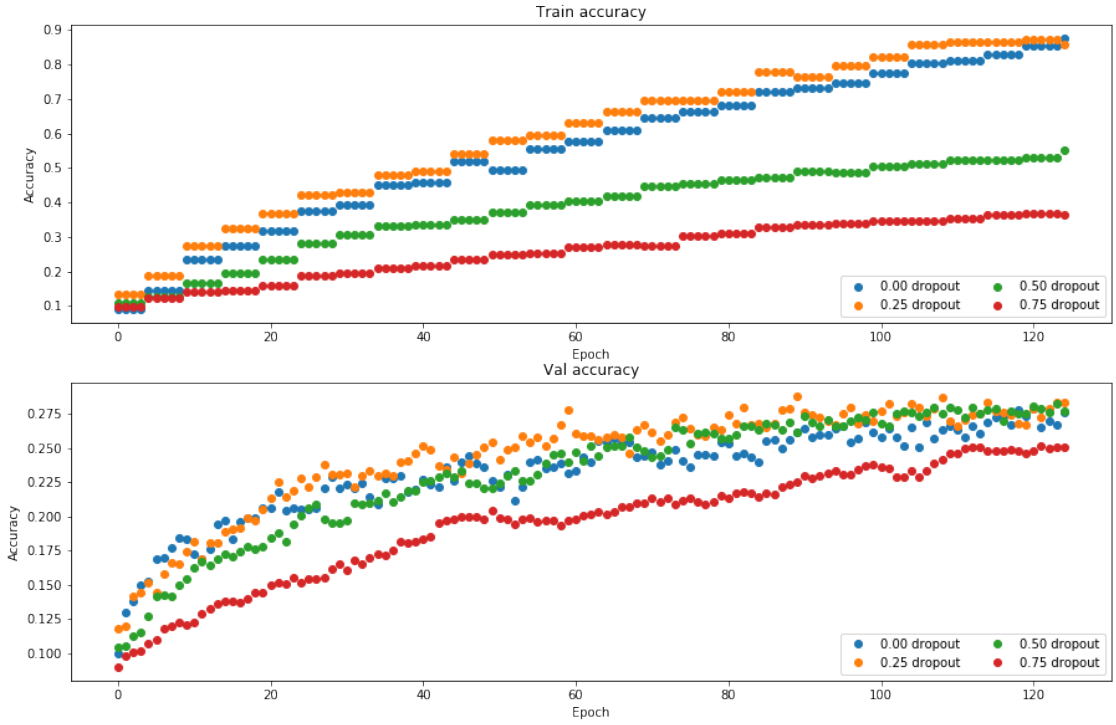
```
for dropout in dropout_choices:
    model = FullyConnectedNet([500], dropout=dropout)
    print(dropout)

    solver = Solver(model, small_data,
                    num_epochs=25, batch_size=100,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-4,
                    },
                    verbose=True, print_every=10)
    solver.train()
    solvers[dropout] = solver
    print('best_val_acc: ', solver.best_val_acc)
```

```
0
(Epoch 0 / 25) (Iteration 1 / 125) loss: 8.596245 train acc: 0.090000 val_acc: 0.100000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 4.380797 train acc: 0.236000 val_acc: 0.173000
(Epoch 4 / 25) (Iteration 21 / 125) loss: 3.113095 train acc: 0.318000 val_acc: 0.206000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 2.451920 train acc: 0.392000 val_acc: 0.223000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 1.830762 train acc: 0.458000 val_acc: 0.227000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 1.836596 train acc: 0.494000 val_acc: 0.222000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 1.673842 train acc: 0.578000 val_acc: 0.233000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 1.414557 train acc: 0.646000 val_acc: 0.247000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 1.025554 train acc: 0.682000 val_acc: 0.253000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 0.701495 train acc: 0.732000 val_acc: 0.264000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 0.679101 train acc: 0.776000 val_acc: 0.258000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 0.829515 train acc: 0.812000 val_acc: 0.263000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 0.527989 train acc: 0.854000 val_acc: 0.279000
best_val_acc:  0.279
0.25
(Epoch 0 / 25) (Iteration 1 / 125) loss: 15.013162 train acc: 0.132000 val_acc: 0.118000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 9.977997 train acc: 0.274000 val_acc: 0.182000
(Epoch 4 / 25) (Iteration 21 / 125) loss: 7.086774 train acc: 0.368000 val_acc: 0.213000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 5.955912 train acc: 0.428000 val_acc: 0.232000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 5.359947 train acc: 0.492000 val_acc: 0.252000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 4.418097 train acc: 0.580000 val_acc: 0.242000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 4.496745 train acc: 0.630000 val_acc: 0.261000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 2.853175 train acc: 0.696000 val_acc: 0.262000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 2.787525 train acc: 0.722000 val_acc: 0.274000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 2.744031 train acc: 0.766000 val_acc: 0.276000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 3.446910 train acc: 0.824000 val_acc: 0.277000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 1.940021 train acc: 0.864000 val_acc: 0.266000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 1.985265 train acc: 0.874000 val_acc: 0.279000
best_val_acc:  0.288
0.5
(Epoch 0 / 25) (Iteration 1 / 125) loss: 5.894457 train acc: 0.108000 val_acc: 0.104000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 4.735179 train acc: 0.166000 val_acc: 0.163000
```

```
(Epoch 4 / 25) (Iteration 21 / 125) loss: 4.528195 train acc: 0.234000 val_acc: 0.184000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 4.268776 train acc: 0.308000 val_acc: 0.197000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 4.150951 train acc: 0.336000 val_acc: 0.225000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 3.789351 train acc: 0.370000 val_acc: 0.224000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 2.854782 train acc: 0.404000 val_acc: 0.247000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 3.119011 train acc: 0.446000 val_acc: 0.243000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 2.270883 train acc: 0.466000 val_acc: 0.257000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 2.907006 train acc: 0.492000 val_acc: 0.273000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 2.590037 train acc: 0.504000 val_acc: 0.268000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 2.967521 train acc: 0.524000 val_acc: 0.278000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 2.645898 train acc: 0.530000 val_acc: 0.281000
best_val_acc:  0.282
0.75
(Epoch 0 / 25) (Iteration 1 / 125) loss: 4.490262 train acc: 0.098000 val_acc: 0.090000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 4.259739 train acc: 0.142000 val_acc: 0.123000
(Epoch 4 / 25) (Iteration 21 / 125) loss: 4.285109 train acc: 0.158000 val_acc: 0.150000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 3.958256 train acc: 0.194000 val_acc: 0.161000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 3.506337 train acc: 0.216000 val_acc: 0.183000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 3.520740 train acc: 0.250000 val_acc: 0.199000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 3.206661 train acc: 0.270000 val_acc: 0.198000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 2.817661 train acc: 0.276000 val_acc: 0.213000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 3.201344 train acc: 0.312000 val_acc: 0.213000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 3.138373 train acc: 0.334000 val_acc: 0.230000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 2.909093 train acc: 0.346000 val_acc: 0.236000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 3.537474 train acc: 0.354000 val_acc: 0.246000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 2.765806 train acc: 0.368000 val_acc: 0.248000
best_val_acc:  0.252
```

In [14]:
```
# 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()
```



6 Question

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

7 Answer

Dropout act as a regularizer which prevents overfitting. At 0 dropout we can see that the training accuracy is very high, but at test time, it is actually (slightly) lower than with dropout. However, at high value of drop out (e.g. 0.75) we can see that both training and test time accuracy are low as the model is now underfitted