2 - Dropout

March 20, 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 [3]: 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:  9.999928899299215
Mean of train-time output:  9.985732280549334
Mean of test-time output:  9.999928899299215
Fraction of train-time output set to zero:  0.30088
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.6
Mean of input:  9.999928899299215
Mean of train-time output:  9.999393251572295
Mean of test-time output:  9.999928899299215
Fraction of train-time output set to zero:  0.599944
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.75
Mean of input:  9.999928899299215
Mean of train-time output:  9.980697647745343
Mean of test-time output:  9.999928899299215
Fraction of train-time output set to zero:  0.750536
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 [4]: 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:  1.8929044903032946e-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 [5]: 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:  2.29898614757146
W1 relative error: 9.74e-07
W2 relative error: 2.43e-08
W3 relative error: 3.04e-08
b1 relative error: 2.01e-08
b2 relative error: 1.90e-09
b3 relative error: 1.30e-10
Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.55e-08
W2 relative error: 2.97e-08
W3 relative error: 4.34e-07
b1 relative error: 1.87e-08
b2 relative error: 5.05e-09
b3 relative error: 7.49e-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 [6]: # 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]
```

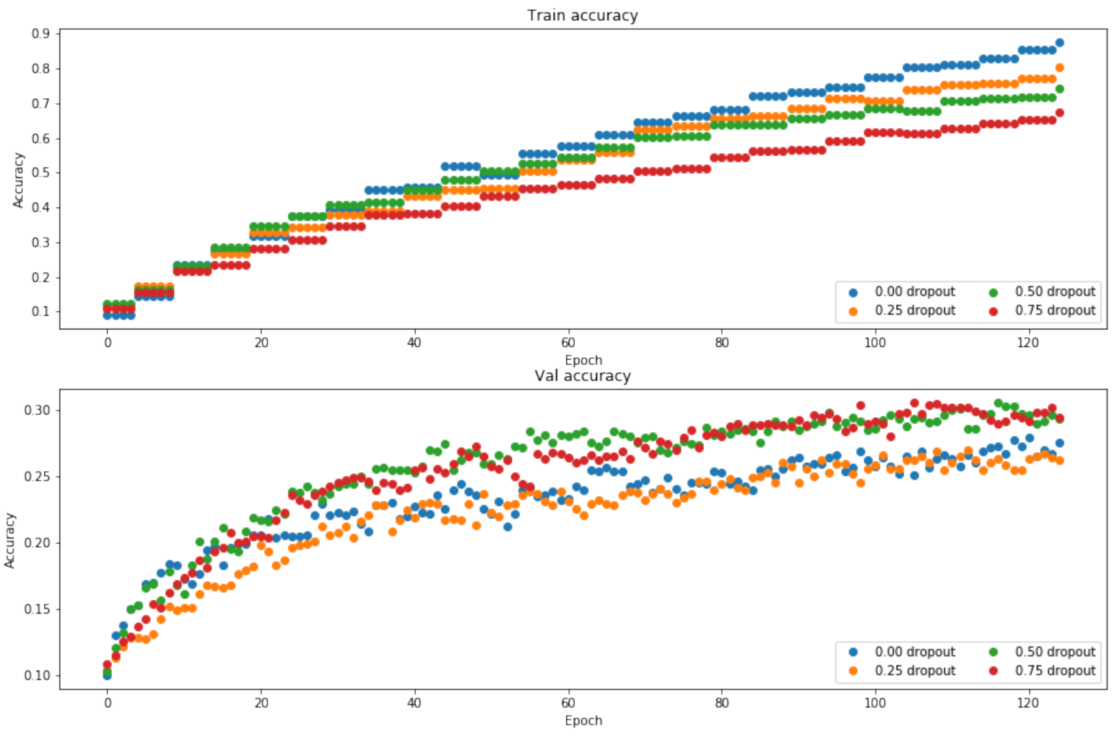
```python
        for dropout in dropout_choices:
            model = FullyConnectedNet([500], dropout=dropout)
            print(dropout)

            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: 10.053351 train acc: 0.114000 val_acc: 0.104000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 5.541544 train acc: 0.226000 val_acc: 0.151000
(Epoch 4 / 25) (Iteration 21 / 125) loss: 4.861298 train acc: 0.328000 val_acc: 0.198000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 3.941271 train acc: 0.378000 val_acc: 0.208000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 3.267835 train acc: 0.432000 val_acc: 0.219000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 3.548183 train acc: 0.454000 val_acc: 0.223000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 3.062070 train acc: 0.538000 val_acc: 0.231000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 2.628531 train acc: 0.624000 val_acc: 0.232000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 2.377977 train acc: 0.658000 val_acc: 0.244000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 2.541462 train acc: 0.684000 val_acc: 0.245000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 1.602210 train acc: 0.706000 val_acc: 0.259000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 1.804731 train acc: 0.754000 val_acc: 0.261000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 1.403584 train acc: 0.770000 val_acc: 0.263000
best_val_acc:  0.27
0.5
(Epoch 0 / 25) (Iteration 1 / 125) loss: 12.066234 train acc: 0.124000 val_acc: 0.103000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 7.516849 train acc: 0.230000 val_acc: 0.161000
```

```
(Epoch 4 / 25) (Iteration 21 / 125) loss: 7.689068 train acc: 0.346000 val_acc: 0.217000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 5.008745 train acc: 0.408000 val_acc: 0.243000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 4.783918 train acc: 0.450000 val_acc: 0.253000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 5.172613 train acc: 0.506000 val_acc: 0.260000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 5.442147 train acc: 0.546000 val_acc: 0.280000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 2.987236 train acc: 0.602000 val_acc: 0.279000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 4.399650 train acc: 0.638000 val_acc: 0.284000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 3.888825 train acc: 0.656000 val_acc: 0.285000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 2.817025 train acc: 0.684000 val_acc: 0.286000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 3.021163 train acc: 0.708000 val_acc: 0.300000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 2.376703 train acc: 0.716000 val_acc: 0.296000
best_val_acc:  0.306
0.75
(Epoch 0 / 25) (Iteration 1 / 125) loss: 15.781663 train acc: 0.110000 val_acc: 0.108000
(Epoch 2 / 25) (Iteration 11 / 125) loss: 11.879495 train acc: 0.216000 val_acc: 0.174000
(Epoch 4 / 25) (Iteration 21 / 125) loss: 11.233345 train acc: 0.282000 val_acc: 0.205000
(Epoch 6 / 25) (Iteration 31 / 125) loss: 9.000759 train acc: 0.346000 val_acc: 0.245000
(Epoch 8 / 25) (Iteration 41 / 125) loss: 8.149151 train acc: 0.384000 val_acc: 0.255000
(Epoch 10 / 25) (Iteration 51 / 125) loss: 7.549874 train acc: 0.432000 val_acc: 0.258000
(Epoch 12 / 25) (Iteration 61 / 125) loss: 6.161986 train acc: 0.466000 val_acc: 0.264000
(Epoch 14 / 25) (Iteration 71 / 125) loss: 6.806906 train acc: 0.506000 val_acc: 0.272000
(Epoch 16 / 25) (Iteration 81 / 125) loss: 6.371577 train acc: 0.544000 val_acc: 0.280000
(Epoch 18 / 25) (Iteration 91 / 125) loss: 4.597612 train acc: 0.568000 val_acc: 0.293000
(Epoch 20 / 25) (Iteration 101 / 125) loss: 5.144893 train acc: 0.618000 val_acc: 0.292000
(Epoch 22 / 25) (Iteration 111 / 125) loss: 5.451678 train acc: 0.628000 val_acc: 0.302000
(Epoch 24 / 25) (Iteration 121 / 125) loss: 5.269128 train acc: 0.652000 val_acc: 0.292000
best_val_acc:  0.306
```

```
In [7]: # 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

No dropout, the final training accuracy is 85% and the validation accuracy is 28%. With 0.25 dropout, the final training accuracy is 77% and the validation accuracy is 26%. With 0.5 dropout, the final training accuracy is 72% and the validation accuracy 30%. With 0.75 dropout, the final training accuracy is 65% and the validation accuracy is 30%. As we can see from the result, the difference between training and validation accuracy has been reduced. So dropout can relieve the problem of overfitting.