

# Dropout

February 12, 2020

## 1 Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and achieve 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](http://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 cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_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-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,)

```

## 1.1 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 [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())

```

```

Running tests with p = 0.3
Mean of input: 9.997243690342161
Mean of train-time output: 10.051213555038881
Mean of test-time output: 9.997243690342161
Fraction of train-time output set to zero: 0.698476
Fraction of test-time output set to zero: 0.0
Running tests with p = 0.6
Mean of input: 9.997243690342161
Mean of train-time output: 9.997800418539825
Mean of test-time output: 9.997243690342161
Fraction of train-time output set to zero: 0.400136
Fraction of test-time output set to zero: 0.0
Running tests with p = 0.75
Mean of input: 9.997243690342161
Mean of train-time output: 9.990889323112652
Mean of test-time output: 9.997243690342161
Fraction of train-time output set to zero: 0.250436
Fraction of test-time output set to zero: 0.0

```

## 1.2 Dropout backward pass

Implement the backward pass, `dropout_backward`, in `nndl/layers.py`. After that, test your gradients by running the following cell:

```
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],
                                                x, dx)

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

dx relative error: 5.4456117023484937e-11
```

## 1.3 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  $W_1$  gradient relative error is on the order of  $1e-6$  (the largest of all the relative errors).

```
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 [0.5, 0.75, 1.0]:
            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 = 0.5
Initial loss: 2.309771209610118
W1 relative error: 2.694274363733021e-07
W2 relative error: 7.439246147919978e-08
W3 relative error: 1.910371122296728e-08
b1 relative error: 4.112891126518e-09
b2 relative error: 5.756217724722137e-10
b3 relative error: 1.3204470857080166e-10
```

```
Running check with dropout = 0.75
Initial loss: 2.306133548427975
W1 relative error: 8.72986097970181e-08
W2 relative error: 2.9777307885797295e-07
W3 relative error: 1.8832780806174298e-08
b1 relative error: 5.379486003985169e-08
b2 relative error: 3.6529949080385546e-09
b3 relative error: 9.987242764516995e-11
```

```
Running check with dropout = 1.0
Initial loss: 2.3053332250963194
W1 relative error: 1.2744095365229032e-06
W2 relative error: 4.678743300473988e-07
W3 relative error: 4.331673892536035e-08
b1 relative error: 4.0853539035931665e-08
b2 relative error: 1.951342257912746e-09
b3 relative error: 9.387142701440351e-11
```

## 1.4 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.

```
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 = [0.6, 1.0]
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',
                    optim_config={
                        'learning_rate': 5e-4,
                    },
                    verbose=True, print_every=100)

    solver.train()
    solvers[dropout] = solver

(Iteration 1 / 125) loss: 2.300199
(Epoch 0 / 25) train acc: 0.158000; val_acc: 0.127000
(Epoch 1 / 25) train acc: 0.132000; val_acc: 0.121000
(Epoch 2 / 25) train acc: 0.204000; val_acc: 0.170000
(Epoch 3 / 25) train acc: 0.240000; val_acc: 0.192000
(Epoch 4 / 25) train acc: 0.312000; val_acc: 0.274000
(Epoch 5 / 25) train acc: 0.314000; val_acc: 0.269000
(Epoch 6 / 25) train acc: 0.364000; val_acc: 0.252000
(Epoch 7 / 25) train acc: 0.390000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.386000; val_acc: 0.290000
(Epoch 9 / 25) train acc: 0.372000; val_acc: 0.267000
(Epoch 10 / 25) train acc: 0.424000; val_acc: 0.286000
(Epoch 11 / 25) train acc: 0.396000; val_acc: 0.275000
(Epoch 12 / 25) train acc: 0.458000; val_acc: 0.299000
(Epoch 13 / 25) train acc: 0.496000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.492000; val_acc: 0.299000
(Epoch 15 / 25) train acc: 0.550000; val_acc: 0.296000
(Epoch 16 / 25) train acc: 0.584000; val_acc: 0.297000
(Epoch 17 / 25) train acc: 0.582000; val_acc: 0.309000
(Epoch 18 / 25) train acc: 0.612000; val_acc: 0.306000
(Epoch 19 / 25) train acc: 0.628000; val_acc: 0.323000
(Epoch 20 / 25) train acc: 0.608000; val_acc: 0.324000
(Iteration 101 / 125) loss: 1.369535
(Epoch 21 / 25) train acc: 0.644000; val_acc: 0.330000
(Epoch 22 / 25) train acc: 0.708000; val_acc: 0.341000
(Epoch 23 / 25) train acc: 0.690000; val_acc: 0.297000
(Epoch 24 / 25) train acc: 0.740000; val_acc: 0.300000
(Epoch 25 / 25) train acc: 0.750000; val_acc: 0.329000
(Iteration 1 / 125) loss: 2.300607
(Epoch 0 / 25) train acc: 0.172000; val_acc: 0.167000
(Epoch 1 / 25) train acc: 0.210000; val_acc: 0.197000
(Epoch 2 / 25) train acc: 0.284000; val_acc: 0.240000
(Epoch 3 / 25) train acc: 0.302000; val_acc: 0.246000
(Epoch 4 / 25) train acc: 0.392000; val_acc: 0.289000

```

```

(Epoch 5 / 25) train acc: 0.420000; val_acc: 0.274000
(Epoch 6 / 25) train acc: 0.420000; val_acc: 0.304000
(Epoch 7 / 25) train acc: 0.474000; val_acc: 0.293000
(Epoch 8 / 25) train acc: 0.516000; val_acc: 0.330000
(Epoch 9 / 25) train acc: 0.566000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.620000; val_acc: 0.321000
(Epoch 11 / 25) train acc: 0.656000; val_acc: 0.317000
(Epoch 12 / 25) train acc: 0.676000; val_acc: 0.319000
(Epoch 13 / 25) train acc: 0.680000; val_acc: 0.304000
(Epoch 14 / 25) train acc: 0.752000; val_acc: 0.323000
(Epoch 15 / 25) train acc: 0.802000; val_acc: 0.321000
(Epoch 16 / 25) train acc: 0.804000; val_acc: 0.300000
(Epoch 17 / 25) train acc: 0.868000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.894000; val_acc: 0.298000
(Epoch 19 / 25) train acc: 0.910000; val_acc: 0.282000
(Epoch 20 / 25) train acc: 0.926000; val_acc: 0.316000
(Iteration 101 / 125) loss: 0.245816
(Epoch 21 / 25) train acc: 0.950000; val_acc: 0.282000
(Epoch 22 / 25) train acc: 0.958000; val_acc: 0.292000
(Epoch 23 / 25) train acc: 0.966000; val_acc: 0.307000
(Epoch 24 / 25) train acc: 0.966000; val_acc: 0.285000
(Epoch 25 / 25) train acc: 0.970000; val_acc: 0.284000

```

In [7]: *# Plot train and validation accuracies of the two models*

```

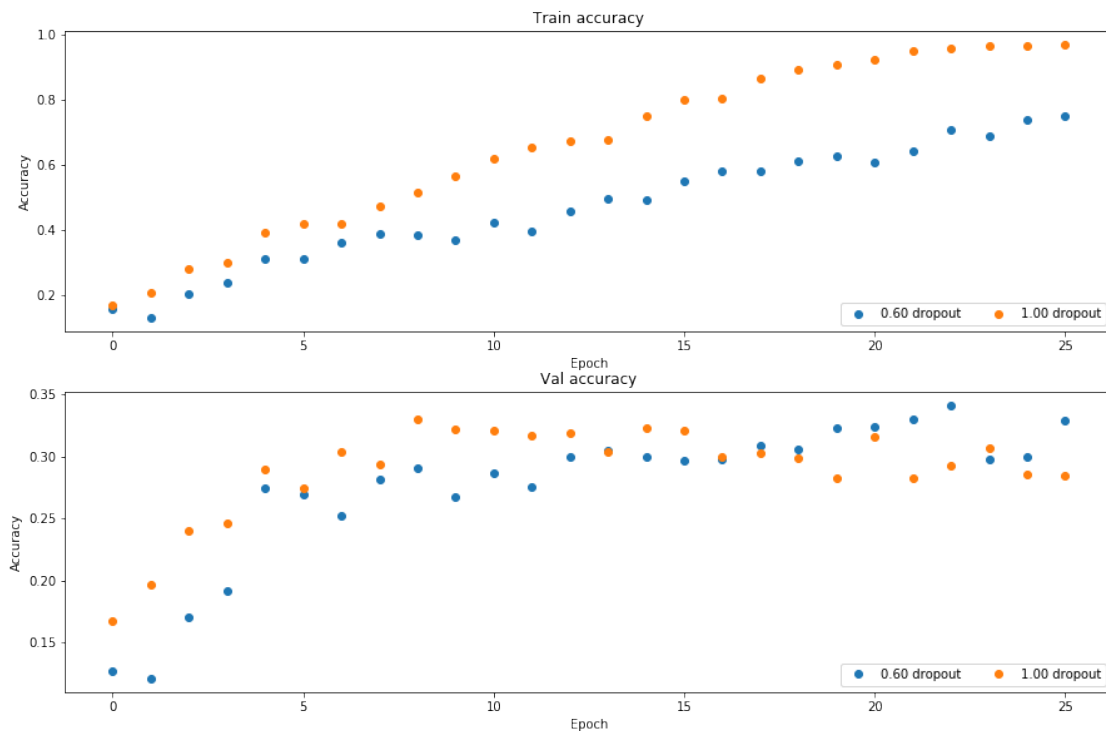
train_accs = []
val_accs = []
for dropout in dropout_choices:
    solver = solvers[dropout]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

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()
```



## 1.5 Question

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

## 1.6 Answer:

Yes, dropout is performing regularization because with dropout, our model has a training accuracy much closer to validation accuracy whereas our model without dropout has a much higher training accuracy than validation accuracy.

**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:

$\min(\text{floor}((X - 32\%)) / 23\%, 1)$  where if you get 55% or higher validation accuracy, you get full points.

```
In [10]: # ===== #
# YOUR CODE HERE:
# Implement a FC-net that achieves at least 55% validation accuracy
# on CIFAR-10.
```

```

# ===== #

optimizer = 'adam'
best_model = None

layer_dims = [500, 500, 500]
weight_scale = 0.01
learning_rate = 5e-3
lr_decay = 0.9
use_batchnorm = True
dropout = 0.5

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale, use_batchnorm=use_batchnorm)

solver = Solver(model, data,
                 num_epochs=10, batch_size=256,
                 update_rule=optimizer,
                 optim_config={
                     'learning_rate': learning_rate,
                 },
                 lr_decay=lr_decay,
                 verbose=True, print_every=50)

solver.train()

# ===== #
# END YOUR CODE HERE
# ===== #

(Iteration 1 / 1910) loss: 2.313253
(Epoch 0 / 10) train acc: 0.156000; val_acc: 0.140000
(Iteration 51 / 1910) loss: 1.900724
(Iteration 101 / 1910) loss: 1.752910
(Iteration 151 / 1910) loss: 1.790791
(Epoch 1 / 10) train acc: 0.434000; val_acc: 0.448000
(Iteration 201 / 1910) loss: 1.627303
(Iteration 251 / 1910) loss: 1.690706
(Iteration 301 / 1910) loss: 1.710692
(Iteration 351 / 1910) loss: 1.608809
(Epoch 2 / 10) train acc: 0.492000; val_acc: 0.486000
(Iteration 401 / 1910) loss: 1.618772
(Iteration 451 / 1910) loss: 1.501616
(Iteration 501 / 1910) loss: 1.434100
(Iteration 551 / 1910) loss: 1.591375
(Epoch 3 / 10) train acc: 0.533000; val_acc: 0.509000
(Iteration 601 / 1910) loss: 1.412635
(Iteration 651 / 1910) loss: 1.277013
(Iteration 701 / 1910) loss: 1.451348
(Iteration 751 / 1910) loss: 1.477529

```



(Epoch 4 / 10) train acc: 0.544000; val\_acc: 0.516000  
(Iteration 801 / 1910) loss: 1.440390  
(Iteration 851 / 1910) loss: 1.421336  
(Iteration 901 / 1910) loss: 1.403937  
(Iteration 951 / 1910) loss: 1.349680  
(Epoch 5 / 10) train acc: 0.536000; val\_acc: 0.549000  
(Iteration 1001 / 1910) loss: 1.371560  
(Iteration 1051 / 1910) loss: 1.296324  
(Iteration 1101 / 1910) loss: 1.444857  
(Epoch 6 / 10) train acc: 0.566000; val\_acc: 0.525000  
(Iteration 1151 / 1910) loss: 1.390857  
(Iteration 1201 / 1910) loss: 1.342500  
(Iteration 1251 / 1910) loss: 1.409043  
(Iteration 1301 / 1910) loss: 1.310190  
(Epoch 7 / 10) train acc: 0.590000; val\_acc: 0.529000  
(Iteration 1351 / 1910) loss: 1.275100  
(Iteration 1401 / 1910) loss: 1.400575  
(Iteration 1451 / 1910) loss: 1.372440  
(Iteration 1501 / 1910) loss: 1.381053  
(Epoch 8 / 10) train acc: 0.603000; val\_acc: 0.546000  
(Iteration 1551 / 1910) loss: 1.342825  
(Iteration 1601 / 1910) loss: 1.404309  
(Iteration 1651 / 1910) loss: 1.274069  
(Iteration 1701 / 1910) loss: 1.350118  
(Epoch 9 / 10) train acc: 0.619000; val\_acc: 0.540000  
(Iteration 1751 / 1910) loss: 1.212386  
(Iteration 1801 / 1910) loss: 1.276342  
(Iteration 1851 / 1910) loss: 1.240554  
(Iteration 1901 / 1910) loss: 1.283522  
(Epoch 10 / 10) train acc: 0.618000; val\_acc: 0.560000