Dropout

February 19, 2024

1 Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive 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).

[3]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from nndl.layers import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from utils.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))))
```

```
The autoreload extension is already loaded. To reload it, use:
```

```
%reload_ext autoreload
```

[4]:
```
# 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.

[5]:
```
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:  9.999809849071406
Mean of train-time output:  9.98637406364458
Mean of test-time output:  9.999809849071406
Fraction of train-time output set to zero:  0.700404
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.999809849071406
Mean of train-time output:  10.016316613580907
Mean of test-time output:  9.999809849071406
Fraction of train-time output set to zero:  0.39916
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.999809849071406
Mean of train-time output:  10.014694269459804
```

```
Mean of test-time output:  9.999809849071406
Fraction of train-time output set to zero:  0.248884
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:

[6]:
```python
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, dout)

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

```
dx relative error:  5.445612258668896e-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 W1 gradient relative error is on the order of 1e-6 (the largest of all the relative errors).

[7]:
```python
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-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num,␣
↪grads[name])))
  print('\n')
```

```
Running check with dropout =  0
Initial loss:  2.3051948273987857
W1 relative error: 2.5272575344376073e-07
W2 relative error: 1.5034484929313676e-05
W3 relative error: 2.753446833630168e-07
b1 relative error: 2.936957476400148e-06
b2 relative error: 5.051339805546953e-08
b3 relative error: 1.1740467838205477e-10


Running check with dropout =  0.25
Initial loss:  2.3126468345657742
W1 relative error: 1.483854795975875e-08
W2 relative error: 2.3427832149940254e-10
W3 relative error: 3.564454999162522e-08
b1 relative error: 1.5292167232408546e-09
b2 relative error: 1.842268868410678e-10
b3 relative error: 8.701800136729388e-11


Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.553387957138422e-08
W2 relative error: 2.974218050584597e-08
W3 relative error: 4.3413247403122424e-07
b1 relative error: 1.872462967441693e-08
b2 relative error: 5.045591219274328e-09
b3 relative error: 7.487013797161614e-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.

[8]:
```
# Train two identical nets, one with dropout and one without

num_train = 500
small_data = {
```

```python
    '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, 0.6]
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
```

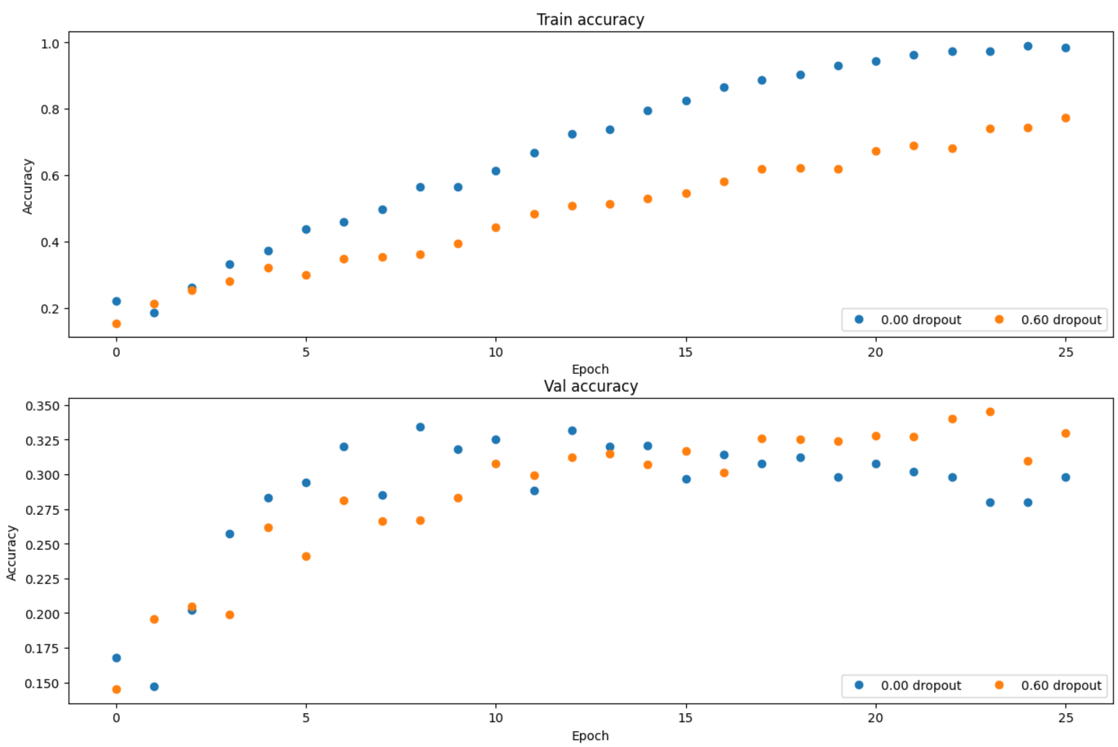
```
(Iteration 1 / 125) loss: 2.300804
(Epoch 0 / 25) train acc: 0.220000; val_acc: 0.168000
(Epoch 1 / 25) train acc: 0.186000; val_acc: 0.147000
(Epoch 2 / 25) train acc: 0.262000; val_acc: 0.202000
(Epoch 3 / 25) train acc: 0.332000; val_acc: 0.257000
(Epoch 4 / 25) train acc: 0.372000; val_acc: 0.283000
(Epoch 5 / 25) train acc: 0.436000; val_acc: 0.294000
(Epoch 6 / 25) train acc: 0.460000; val_acc: 0.320000
(Epoch 7 / 25) train acc: 0.496000; val_acc: 0.285000
(Epoch 8 / 25) train acc: 0.564000; val_acc: 0.334000
(Epoch 9 / 25) train acc: 0.564000; val_acc: 0.318000
(Epoch 10 / 25) train acc: 0.612000; val_acc: 0.325000
(Epoch 11 / 25) train acc: 0.668000; val_acc: 0.288000
(Epoch 12 / 25) train acc: 0.724000; val_acc: 0.332000
(Epoch 13 / 25) train acc: 0.738000; val_acc: 0.320000
(Epoch 14 / 25) train acc: 0.794000; val_acc: 0.321000
(Epoch 15 / 25) train acc: 0.824000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.864000; val_acc: 0.314000
(Epoch 17 / 25) train acc: 0.886000; val_acc: 0.308000
(Epoch 18 / 25) train acc: 0.904000; val_acc: 0.312000
(Epoch 19 / 25) train acc: 0.930000; val_acc: 0.298000
(Epoch 20 / 25) train acc: 0.944000; val_acc: 0.308000
(Iteration 101 / 125) loss: 0.165809
(Epoch 21 / 25) train acc: 0.962000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.974000; val_acc: 0.298000
(Epoch 23 / 25) train acc: 0.974000; val_acc: 0.280000
```

```
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.280000
(Epoch 25 / 25) train acc: 0.984000; val_acc: 0.298000
(Iteration 1 / 125) loss: 2.301328
(Epoch 0 / 25) train acc: 0.154000; val_acc: 0.145000
(Epoch 1 / 25) train acc: 0.212000; val_acc: 0.196000
(Epoch 2 / 25) train acc: 0.252000; val_acc: 0.205000
(Epoch 3 / 25) train acc: 0.280000; val_acc: 0.199000
(Epoch 4 / 25) train acc: 0.322000; val_acc: 0.262000
(Epoch 5 / 25) train acc: 0.298000; val_acc: 0.241000
(Epoch 6 / 25) train acc: 0.348000; val_acc: 0.281000
(Epoch 7 / 25) train acc: 0.352000; val_acc: 0.266000
(Epoch 8 / 25) train acc: 0.362000; val_acc: 0.267000
(Epoch 9 / 25) train acc: 0.394000; val_acc: 0.283000
(Epoch 10 / 25) train acc: 0.444000; val_acc: 0.308000
(Epoch 11 / 25) train acc: 0.484000; val_acc: 0.299000
(Epoch 12 / 25) train acc: 0.508000; val_acc: 0.312000
(Epoch 13 / 25) train acc: 0.512000; val_acc: 0.315000
(Epoch 14 / 25) train acc: 0.528000; val_acc: 0.307000
(Epoch 15 / 25) train acc: 0.546000; val_acc: 0.317000
(Epoch 16 / 25) train acc: 0.582000; val_acc: 0.301000
(Epoch 17 / 25) train acc: 0.618000; val_acc: 0.326000
(Epoch 18 / 25) train acc: 0.620000; val_acc: 0.325000
(Epoch 19 / 25) train acc: 0.618000; val_acc: 0.324000
(Epoch 20 / 25) train acc: 0.674000; val_acc: 0.328000
(Iteration 101 / 125) loss: 1.236431
(Epoch 21 / 25) train acc: 0.690000; val_acc: 0.327000
(Epoch 22 / 25) train acc: 0.682000; val_acc: 0.340000
(Epoch 23 / 25) train acc: 0.740000; val_acc: 0.345000
(Epoch 24 / 25) train acc: 0.744000; val_acc: 0.310000
(Epoch 25 / 25) train acc: 0.774000; val_acc: 0.330000
```

[9]:

```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' %
 ↪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 since we see that the validation accuracies are similar with or without dropout, but the training accuracy is lower in the model with dropout which implies that it regularized overfitting the training data.

1.7 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(floor((X - 32%)) / 23%, 1) where if you get 55% or higher validation accuracy, you get full points.

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

layer_dims = [200, 200, 200]

solvers = {}
model = FullyConnectedNet(layer_dims, weight_scale=5e-2, dropout=0.8,
                          use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=20, batch_size=500,
                update_rule='adam',
                optim_config = {
                  'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                verbose=True, print_every=100)
solver.train()

y_test_pred = np.argmax(model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)
print('Validation set accuracy: {}'.format(np.mean(y_val_pred ==
 ↪data['y_val'])))
print('Test set accuracy: {}'.format(np.mean(y_test_pred == data['y_test'])))

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

```
(Iteration 1 / 1960) loss: 2.398551
(Epoch 0 / 20) train acc: 0.193000; val_acc: 0.180000
(Epoch 1 / 20) train acc: 0.472000; val_acc: 0.457000
(Iteration 101 / 1960) loss: 1.533899
(Epoch 2 / 20) train acc: 0.491000; val_acc: 0.477000
(Iteration 201 / 1960) loss: 1.500319
(Epoch 3 / 20) train acc: 0.521000; val_acc: 0.519000
(Iteration 301 / 1960) loss: 1.422592
(Epoch 4 / 20) train acc: 0.545000; val_acc: 0.529000
```

```
(Iteration 401 / 1960) loss: 1.357056
(Epoch 5 / 20) train acc: 0.580000; val_acc: 0.552000
(Iteration 501 / 1960) loss: 1.341525
(Epoch 6 / 20) train acc: 0.549000; val_acc: 0.538000
(Iteration 601 / 1960) loss: 1.290470
(Epoch 7 / 20) train acc: 0.590000; val_acc: 0.545000
(Iteration 701 / 1960) loss: 1.236626
(Epoch 8 / 20) train acc: 0.613000; val_acc: 0.544000
(Iteration 801 / 1960) loss: 1.228739
(Epoch 9 / 20) train acc: 0.628000; val_acc: 0.549000
(Iteration 901 / 1960) loss: 1.118770
(Epoch 10 / 20) train acc: 0.631000; val_acc: 0.548000
(Iteration 1001 / 1960) loss: 1.193559
(Epoch 11 / 20) train acc: 0.623000; val_acc: 0.577000
(Iteration 1101 / 1960) loss: 1.179836
(Epoch 12 / 20) train acc: 0.632000; val_acc: 0.556000
(Iteration 1201 / 1960) loss: 1.128166
(Epoch 13 / 20) train acc: 0.633000; val_acc: 0.546000
(Iteration 1301 / 1960) loss: 1.152440
(Epoch 14 / 20) train acc: 0.671000; val_acc: 0.558000
(Iteration 1401 / 1960) loss: 1.186915
(Epoch 15 / 20) train acc: 0.672000; val_acc: 0.559000
(Iteration 1501 / 1960) loss: 1.026532
(Epoch 16 / 20) train acc: 0.687000; val_acc: 0.554000
(Iteration 1601 / 1960) loss: 1.033913
(Epoch 17 / 20) train acc: 0.687000; val_acc: 0.570000
(Iteration 1701 / 1960) loss: 1.022657
(Epoch 18 / 20) train acc: 0.700000; val_acc: 0.556000
(Iteration 1801 / 1960) loss: 1.006586
(Epoch 19 / 20) train acc: 0.724000; val_acc: 0.559000
(Iteration 1901 / 1960) loss: 0.937301
(Epoch 20 / 20) train acc: 0.696000; val_acc: 0.562000
Validation set accuracy: 0.577
Test set accuracy: 0.565
```