Dropout

February 10, 2021

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).

[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))))
```

[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.

[6]:
```
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:  10.002172211000216
Mean of train-time output:  9.976094425529148
Mean of test-time output:  10.002172211000216
Fraction of train-time output set to zero:  0.700784
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.002172211000216
Mean of train-time output:  10.001277696843985
Mean of test-time output:  10.002172211000216
Fraction of train-time output set to zero:  0.399984
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.002172211000216
Mean of train-time output:  9.990919594597642
Mean of test-time output:  10.002172211000216
Fraction of train-time output set to zero:  0.250792
```

```
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:

[7]:
```
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:  5.445609616803788e-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).

[10]:
```
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:  2.2991386421748192
W1 relative error: 7.973832956782614e-07
W2 relative error: 4.158867717441295e-06
W3 relative error: 8.735367521709817e-07
b1 relative error: 1.460431157438847e-08
b2 relative error: 1.7950404736239145e-09
b3 relative error: 1.9021861197309654e-10


Running check with dropout =  0.25
Initial loss:  2.3051082895840453
W1 relative error: 9.159767628234904e-08
W2 relative error: 9.295433636745756e-10
W3 relative error: 8.40729637598436e-09
b1 relative error: 3.73645764917359e-08
b2 relative error: 1.773539976217994e-10
b3 relative error: 1.3790600574173082e-10


Running check with dropout =  0.5
Initial loss:  2.3073848515258604
W1 relative error: 3.235266747322489e-08
W2 relative error: 9.643264450138405e-08
W3 relative error: 4.156906738072875e-07
b1 relative error: 3.6899715043001694e-09
b2 relative error: 3.367109508501977e-09
b3 relative error: 7.496979925192784e-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.

[11]:
```
# 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, 0.6]
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: 0.188000; val_acc: 0.147000
(Epoch 2 / 25) train acc: 0.266000; val_acc: 0.200000
(Epoch 3 / 25) train acc: 0.338000; val_acc: 0.262000
(Epoch 4 / 25) train acc: 0.378000; val_acc: 0.278000
(Epoch 5 / 25) train acc: 0.428000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.468000; val_acc: 0.323000
(Epoch 7 / 25) train acc: 0.494000; val_acc: 0.287000
(Epoch 8 / 25) train acc: 0.566000; val_acc: 0.328000
(Epoch 9 / 25) train acc: 0.572000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.622000; val_acc: 0.324000
(Epoch 11 / 25) train acc: 0.670000; val_acc: 0.279000
(Epoch 12 / 25) train acc: 0.710000; val_acc: 0.338000
(Epoch 13 / 25) train acc: 0.746000; val_acc: 0.319000
(Epoch 14 / 25) train acc: 0.792000; val_acc: 0.307000
(Epoch 15 / 25) train acc: 0.834000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.876000; val_acc: 0.327000
(Epoch 17 / 25) train acc: 0.886000; val_acc: 0.320000
(Epoch 18 / 25) train acc: 0.918000; val_acc: 0.314000
(Epoch 19 / 25) train acc: 0.922000; val_acc: 0.290000
(Epoch 20 / 25) train acc: 0.944000; val_acc: 0.306000
(Iteration 101 / 125) loss: 0.156105
(Epoch 21 / 25) train acc: 0.968000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.302000
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.986000; val_acc: 0.285000
```

```
(Epoch 25 / 25) train acc: 0.978000; val_acc: 0.311000
(Iteration 1 / 125) loss: 2.301328
(Epoch 0 / 25) train acc: 0.154000; val_acc: 0.143000
(Epoch 1 / 25) train acc: 0.214000; val_acc: 0.195000
(Epoch 2 / 25) train acc: 0.252000; val_acc: 0.217000
(Epoch 3 / 25) train acc: 0.276000; val_acc: 0.200000
(Epoch 4 / 25) train acc: 0.308000; val_acc: 0.254000
(Epoch 5 / 25) train acc: 0.316000; val_acc: 0.241000
(Epoch 6 / 25) train acc: 0.322000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.354000; val_acc: 0.273000
(Epoch 8 / 25) train acc: 0.364000; val_acc: 0.276000
(Epoch 9 / 25) train acc: 0.408000; val_acc: 0.282000
(Epoch 10 / 25) train acc: 0.454000; val_acc: 0.302000
(Epoch 11 / 25) train acc: 0.472000; val_acc: 0.297000
(Epoch 12 / 25) train acc: 0.498000; val_acc: 0.318000
(Epoch 13 / 25) train acc: 0.510000; val_acc: 0.309000
(Epoch 14 / 25) train acc: 0.534000; val_acc: 0.315000
(Epoch 15 / 25) train acc: 0.546000; val_acc: 0.331000
(Epoch 16 / 25) train acc: 0.584000; val_acc: 0.302000
(Epoch 17 / 25) train acc: 0.626000; val_acc: 0.332000
(Epoch 18 / 25) train acc: 0.614000; val_acc: 0.327000
(Epoch 19 / 25) train acc: 0.626000; val_acc: 0.325000
(Epoch 20 / 25) train acc: 0.656000; val_acc: 0.338000
(Iteration 101 / 125) loss: 1.299296
(Epoch 21 / 25) train acc: 0.676000; val_acc: 0.329000
(Epoch 22 / 25) train acc: 0.684000; val_acc: 0.325000
(Epoch 23 / 25) train acc: 0.730000; val_acc: 0.343000
(Epoch 24 / 25) train acc: 0.736000; val_acc: 0.321000
(Epoch 25 / 25) train acc: 0.768000; val_acc: 0.333000
```

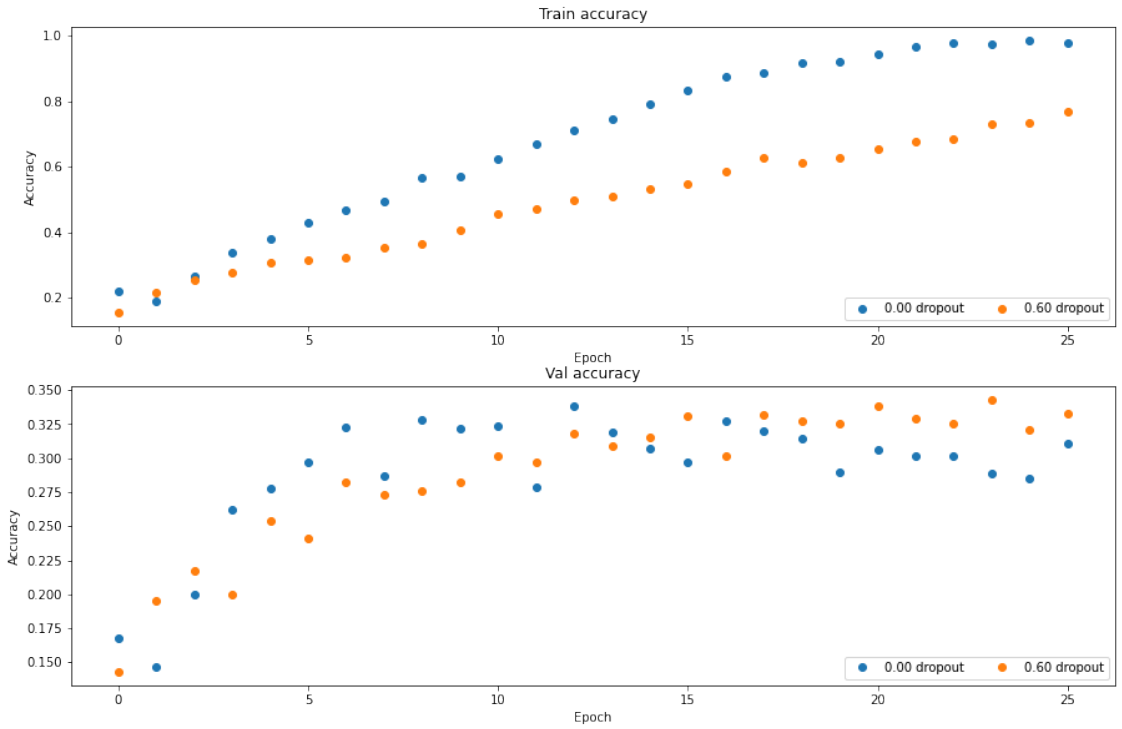
[12]:
```
# Plot train and validation accuracies of the two models

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 performs regularization because the model increases validation accuracy but does not decrease training accuracy, making the model more generalizable. Overfitting can be seen in the model without dropout, as training accuracy is very high at above 90% but the validation accuracy is around 0.3, indicating overfitting. Adding dropout makes the training curve a less steep curve with lower training accuracy in general, but the validation accuracy is higher and indicates less

overfitting.

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%)) / 28%, 1) where if you get 60% or higher validation accuracy, you get full points.

[15]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #

optimizer = 'adam'
dropout = 0.65
hidden_dims = [500, 500, 500]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(hidden_dims,
                          weight_scale=weight_scale,
                          dropout=dropout,
                          use_batchnorm=True)
solver = Solver(model,
                data,
                num_epochs=10,
                batch_size=100,
                update_rule=optimizer,
                optim_config={'learning_rate': learning_rate},
                lr_decay=lr_decay,
                verbose=True,
                print_every=100
               )

solver.train()

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

```
(Iteration 1 / 4900) loss: 2.286936
(Epoch 0 / 10) train acc: 0.194000; val_acc: 0.202000
(Iteration 101 / 4900) loss: 2.015151
(Iteration 201 / 4900) loss: 1.688241
(Iteration 301 / 4900) loss: 1.512466
```

```
(Iteration 401 / 4900) loss: 1.714905
(Epoch 1 / 10) train acc: 0.501000; val_acc: 0.476000
(Iteration 501 / 4900) loss: 1.491691
(Iteration 601 / 4900) loss: 1.727417
(Iteration 701 / 4900) loss: 1.473698
(Iteration 801 / 4900) loss: 1.541122
(Iteration 901 / 4900) loss: 1.463471
(Epoch 2 / 10) train acc: 0.521000; val_acc: 0.497000
(Iteration 1001 / 4900) loss: 1.235988
(Iteration 1101 / 4900) loss: 1.380242
(Iteration 1201 / 4900) loss: 1.311429
(Iteration 1301 / 4900) loss: 1.470639
(Iteration 1401 / 4900) loss: 1.418507
(Epoch 3 / 10) train acc: 0.572000; val_acc: 0.523000
(Iteration 1501 / 4900) loss: 1.575684
(Iteration 1601 / 4900) loss: 1.257184
(Iteration 1701 / 4900) loss: 1.216112
(Iteration 1801 / 4900) loss: 1.310623
(Iteration 1901 / 4900) loss: 1.450710
(Epoch 4 / 10) train acc: 0.556000; val_acc: 0.533000
(Iteration 2001 / 4900) loss: 1.198303
(Iteration 2101 / 4900) loss: 1.357430
(Iteration 2201 / 4900) loss: 1.413133
(Iteration 2301 / 4900) loss: 1.135990
(Iteration 2401 / 4900) loss: 1.307396
(Epoch 5 / 10) train acc: 0.588000; val_acc: 0.545000
(Iteration 2501 / 4900) loss: 1.165337
(Iteration 2601 / 4900) loss: 1.464810
(Iteration 2701 / 4900) loss: 1.170133
(Iteration 2801 / 4900) loss: 1.228508
(Iteration 2901 / 4900) loss: 1.323958
(Epoch 6 / 10) train acc: 0.612000; val_acc: 0.546000
(Iteration 3001 / 4900) loss: 1.305573
(Iteration 3101 / 4900) loss: 1.210016
(Iteration 3201 / 4900) loss: 1.103582
(Iteration 3301 / 4900) loss: 1.161539
(Iteration 3401 / 4900) loss: 1.130704
(Epoch 7 / 10) train acc: 0.628000; val_acc: 0.568000
(Iteration 3501 / 4900) loss: 1.127175
(Iteration 3601 / 4900) loss: 1.074670
(Iteration 3701 / 4900) loss: 1.203933
(Iteration 3801 / 4900) loss: 1.248469
(Iteration 3901 / 4900) loss: 1.230305
(Epoch 8 / 10) train acc: 0.643000; val_acc: 0.569000
(Iteration 4001 / 4900) loss: 1.140865
(Iteration 4101 / 4900) loss: 1.199728
(Iteration 4201 / 4900) loss: 1.173846
(Iteration 4301 / 4900) loss: 1.009089
```

```
(Iteration 4401 / 4900) loss: 0.972446
(Epoch 9 / 10) train acc: 0.668000; val_acc: 0.560000
(Iteration 4501 / 4900) loss: 1.119916
(Iteration 4601 / 4900) loss: 1.086789
(Iteration 4701 / 4900) loss: 1.268793
(Iteration 4801 / 4900) loss: 1.358139
(Epoch 10 / 10) train acc: 0.658000; val_acc: 0.563000
```

2 layers.py

[16]:
```python
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

def affine_forward(x, w, b):
  """
  Computes the forward pass for an affine (fully-connected) layer.

  The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
  examples, where each example x[i] has shape (d_1, ..., d_k). We will
  reshape each input into a vector of dimension D = d_1 * ... * d_k, and
  then transform it to an output vector of dimension M.

  Inputs:
  - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
  - w: A numpy array of weights, of shape (D, M)
  - b: A numpy array of biases, of shape (M,)

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
```

```
  # ============================================================ #

  x_reshaped = x.reshape(x.shape[0], np.prod(x.shape[1:]))
  out = np.dot(x_reshaped, w) + b

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

  cache = (x, w, b)
  return out, cache


def affine_backward(dout, cache):
  """
  Computes the backward pass for an affine layer.

  Inputs:
  - dout: Upstream derivative, of shape (N, M)
  - cache: Tuple of:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

  Returns a tuple of:
  - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
  - dw: Gradient with respect to w, of shape (D, M)
  - db: Gradient with respect to b, of shape (M,)
  """
  x, w, b = cache
  dx, dw, db = None, None, None

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the gradients for the backward pass.
  # Notice:
  #   dout is N x M
  #   dx should be N x d1 x ... x dk; it relates to dout through multiplication␣
 ↪with w, which is D x M
  #   dw should be D x M; it relates to dout through multiplication with x,␣
 ↪which is N x D after reshaping
  #   db should be M; it is just the sum over dout examples
  # ============================================================ #

  N = dout.shape[0] # 10 , also equal to x.shape[0]
  M = dout.shape[1] # 5
  D = np.prod(x.shape[1:]) # 6
```

```
  dx = np.dot(dout, w.T).reshape(x.shape) # (N,M) * (M,D) = (N,D) = (10,6) ->␣
↪reshape to (10, 2, 3)
  dw = np.dot(dout.T, x.reshape(N, D)).T # ((M,N) * (N,D)).T = (D,M) = (6,5)
  db = np.sum(dout, axis=0) # sum down columns/examples of (N,M) matrix -> (M,)␣
↪= (5,)

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

  return dx, dw, db

def relu_forward(x):
  """
  Computes the forward pass for a layer of rectified linear units (ReLUs).

  Input:
  - x: Inputs, of any shape

  Returns a tuple of:
  - out: Output, of the same shape as x
  - cache: x
  """
  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU forward pass.
  # ============================================================ #

  out = np.copy(x)
  out[out <= 0] = 0

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

  cache = x
  return out, cache


def relu_backward(dout, cache):
  """
  Computes the backward pass for a layer of rectified linear units (ReLUs).

  Input:
  - dout: Upstream derivatives, of any shape
  - cache: Input x, of same shape as dout
```

```
  Returns:
  - dx: Gradient with respect to x
  """
  x = cache

  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ============================================================ #

  # ReLU directs linearly to those > 0
  x[x<=0] = 0
  x[x>0] = 1
  dx = np.multiply(dout, x)

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

  return dx

def batchnorm_forward(x, gamma, beta, bn_param):
  """
  Forward pass for batch normalization.

  During training the sample mean and (uncorrected) sample variance are
  computed from minibatch statistics and used to normalize the incoming data.
  During training we also keep an exponentially decaying running mean of the␣
 ↪mean
  and variance of each feature, and these averages are used to normalize data
  at test-time.

  At each timestep we update the running averages for mean and variance using
  an exponential decay based on the momentum parameter:

  running_mean = momentum * running_mean + (1 - momentum) * sample_mean
  running_var = momentum * running_var + (1 - momentum) * sample_var

  Note that the batch normalization paper suggests a different test-time
  behavior: they compute sample mean and variance for each feature using a
  large number of training images rather than using a running average. For
  this implementation we have chosen to use running averages instead since
  they do not require an additional estimation step; the torch7 implementation
  of batch normalization also uses running averages.

  Input:
```

```
  - x: Data of shape (N, D)
  - gamma: Scale parameter of shape (D,)
  - beta: Shift paremeter of shape (D,)
  - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance.
    - running_mean: Array of shape (D,) giving running mean of features
    - running_var Array of shape (D,) giving running variance of features

  Returns a tuple of:
  - out: of shape (N, D)
  - cache: A tuple of values needed in the backward pass
  """
  mode = bn_param['mode']
  eps = bn_param.get('eps', 1e-5)
  momentum = bn_param.get('momentum', 0.9)

  N, D = x.shape
  running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
  running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

  out, cache = None, None
  if mode == 'train':

    # ============================================================ #
    # YOUR CODE HERE:
    #   A few steps here:
    #     (1) Calculate the running mean and variance of the minibatch.
    #     (2) Normalize the activations with the running mean and variance.
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass in
    #         the 'cache' variable.
    # ============================================================ #

    sample_mean = np.mean(x, axis=0)
    sample_var = np.mean(np.square(x - sample_mean), axis=0)

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    x_hat = (x-sample_mean) / np.sqrt(sample_var+eps)
    out = gamma * x_hat + beta

    cache = (x_hat, x, gamma, eps, sample_mean, sample_var)
```

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

  elif mode == 'test':

    # ============================================================ #
    # YOUR CODE HERE:
    #   Calculate the testing time normalized activation.  Normalize using
    #   the running mean and variance, and then scale and shift appropriately.
    #   Store the output as 'out'.
    # ============================================================ #

    x_hat = (x - running_mean) / np.sqrt(running_var + eps)
    out = gamma * x_hat + beta

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

  else:
    raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

  # Store the updated running means back into bn_param
  bn_param['running_mean'] = running_mean
  bn_param['running_var'] = running_var

  return out, cache

def batchnorm_backward(dout, cache):
  """
  Backward pass for batch normalization.

  For this implementation, you should write out a computation graph for
  batch normalization on paper and propagate gradients backward through
  intermediate nodes.

  Inputs:
  - dout: Upstream derivatives, of shape (N, D)
  - cache: Variable of intermediates from batchnorm_forward.

  Returns a tuple of:
  - dx: Gradient with respect to inputs x, of shape (N, D)
  - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
  - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
  """
  dx, dgamma, dbeta = None, None, None
```

```
  # ============================================================== #
  # YOUR CODE HERE:
  #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
  # ============================================================== #

  N = dout.shape[0]
  x_hat, x, gamma, eps, mean, var = cache
  sqrt_var_plus_eps_inv = 1 / np.sqrt(var+eps)
  x_minus_mean = x - mean

  dgamma = np.sum(np.multiply(dout, x_hat), axis=0)
  dbeta = np.sum(dout, axis=0)

  # dx = dl_da + ((2*(x - mean)/N) * dl_dvar) + ((1/N) * dl_dmean)
  dl_dxhat = dout * gamma
  dl_da = sqrt_var_plus_eps_inv * dl_dxhat
  dl_de = (1/2) * sqrt_var_plus_eps_inv * -(sqrt_var_plus_eps_inv ** 2) *␣
↪x_minus_mean * dl_dxhat
  dl_dvar = np.sum(dl_de, axis=0)

  dl_dmean = -sqrt_var_plus_eps_inv * np.sum(dl_dxhat, axis=0) - dl_dvar * (2/
↪N) * np.sum(x_minus_mean, axis=0)

  dx = dl_da + (2* x_minus_mean / N) * dl_dvar + (dl_dmean/N)

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

  return dx, dgamma, dbeta

def dropout_forward(x, dropout_param):
  """
  Performs the forward pass for (inverted) dropout.

  Inputs:
  - x: Input data, of any shape
  - dropout_param: A dictionary with the following keys:
    - p: Dropout parameter. We keep each neuron output with probability p.
    - mode: 'test' or 'train'. If the mode is train, then perform dropout;
      if the mode is test, then just return the input.
    - seed: Seed for the random number generator. Passing seed makes this
      function deterministic, which is needed for gradient checking but not in
      real networks.

  Outputs:
```

```
  - out: Array of the same shape as x.
  - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
    mask that was used to multiply the input; in test mode, mask is None.
  """
  p, mode = dropout_param['p'], dropout_param['mode']
  if 'seed' in dropout_param:
    np.random.seed(dropout_param['seed'])

  mask = None
  out = None

  if mode == 'train':
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during training time.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ============================================================ #

    mask = (np.random.rand(x.shape[0], x.shape[1]) < p) / p
    out = np.multiply(x, mask)

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

  elif mode == 'test':

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout forward pass during test time.
    # ============================================================ #

    out = x

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

  cache = (dropout_param, mask)
  out = out.astype(x.dtype, copy=False)

  return out, cache

def dropout_backward(dout, cache):
  """
  Perform the backward pass for (inverted) dropout.
```

```
  Inputs:
  - dout: Upstream derivatives, of any shape
  - cache: (dropout_param, mask) from dropout_forward.
  """
  dropout_param, mask = cache
  mode = dropout_param['mode']

  dx = None
  if mode == 'train':
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during training time.
    # ============================================================ #

    dx = np.multiply(dout, mask)

    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
  elif mode == 'test':
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during test time.
    # ============================================================ #

    dx = dout

    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
  return dx

def svm_loss(x, y):
  """
  Computes the loss and gradient using for multiclass SVM classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """
```

```
  N = x.shape[0]
  correct_class_scores = x[np.arange(N), y]
  margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
  margins[np.arange(N), y] = 0
  loss = np.sum(margins) / N
  num_pos = np.sum(margins > 0, axis=1)
  dx = np.zeros_like(x)
  dx[margins > 0] = 1
  dx[np.arange(N), y] -= num_pos
  dx /= N
  return loss, dx


def softmax_loss(x, y):
  """
  Computes the loss and gradient for softmax classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """

  probs = np.exp(x - np.max(x, axis=1, keepdims=True))
  probs /= np.sum(probs, axis=1, keepdims=True)
  N = x.shape[0]
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
  dx = probs.copy()
  dx[np.arange(N), y] -= 1
  dx /= N
  return loss, dx
```

3 fc_net.py

[ ]:
```
import numpy as np

from .layers import *
from .layer_utils import *

"""
This code was originally written for CS 231n at Stanford University
```

```
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network with ReLU nonlinearity and
  softmax loss that uses a modular layer design. We assume an input dimension
  of D, a hidden dimension of H, and perform classification over C classes.

  The architecure should be affine - relu - affine - softmax.

  Note that this class does not implement gradient descent; instead, it
  will interact with a separate Solver object that is responsible for running
  optimization.

  The learnable parameters of the model are stored in the dictionary
  self.params that maps parameter names to numpy arrays.
  """

  def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
               dropout=0, weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.

    Inputs:
    - input_dim: An integer giving the size of the input
    - hidden_dims: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
    """
    self.params = {}
    self.reg = reg

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
```

```
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ============================================================ #

    # randn gives distribution from standardized normal distribution with mean␣
↪0 and variance 1
    self.params['W1'] = np.random.normal(loc=0, scale=weight_scale,␣
↪size=(input_dim, hidden_dims))
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = np.random.normal(loc=0, scale=weight_scale,␣
↪size=(hidden_dims, num_classes))
    self.params['b2'] = np.zeros(num_classes)

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

  def loss(self, X, y=None):
    """
    Compute loss and gradient for a minibatch of data.

    Inputs:
    - X: Array of input data of shape (N, d_1, ..., d_k)
    - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

    Returns:
    If y is None, then run a test-time forward pass of the model and return:
    - scores: Array of shape (N, C) giving classification scores, where
      scores[i, c] is the classification score for X[i] and class c.

    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ============================================================ #

    # cache: (x, w, b)
```

```
    out_affine_1_relu, cache_affine_1_relu = affine_relu_forward(X, self.
↪params['W1'], self.params['b1'])
    scores, cache_affine_2 = affine_forward(out_affine_1_relu, self.
↪params['W2'], self.params['b2'])

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

    # If y is None then we are in test mode so just return scores
    if y is None:
      return scores

    loss, grads = 0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Store
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ============================================================ #

    loss, dL = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(np.square(self.params['W1'])) + np.sum(np.
↪square(self.params['W2'])))

    dH, grads['W2'], grads['b2'] = affine_backward(dL, cache_affine_2)
    _, grads['W1'], grads['b1'] = affine_relu_backward(dH, cache_affine_1_relu)

    grads['W2'] += self.reg * self.params['W2'] # d(0.5 * reg * (W1**2 +␣
↪W2**2)) / d(W2) = reg * W2
    grads['W1'] += self.reg * self.params['W1'] # d(0.5 * reg * (W1**2 +␣
↪W2**2)) / d(W1) = reg * W1



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

```
    return loss, grads


class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden layer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify.
    - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch normalization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be performed using
      this datatype. float32 is faster but less accurate, so you should use
      float64 for numeric gradient checking.
    - seed: If not None, then pass this random seed to the dropout layers. This
      will make the dropout layers deteriminstic so we can gradient check the
      model.
    """
    self.use_batchnorm = use_batchnorm
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len(hidden_dims)
    self.dtype = dtype
```

```
    self.params = {}

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params dictionary.
    #   The weights and biases of layer 1 are W1 and b1; and in general the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    # ============================================================ #

    dims = [input_dim] + hidden_dims + [num_classes]
    for i in range(self.num_layers):
        digit = str(i+1)
        self.params['W' + digit] = np.random.normal(0, weight_scale, (dims[i],␣
↪dims[i+1]))
        self.params['b' + digit] = np.zeros(dims[i+1])
        if self.use_batchnorm:
            if i + 1 == self.num_layers:
                break
            self.params['gamma' + digit] = np.ones(dims[i+1])
            self.params['beta' + digit] = np.zeros(dims[i+1])

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

    # When using dropout we need to pass a dropout_param dictionary to each
    # dropout layer so that the layer knows the dropout probability and the mode
    # (train / test). You can pass the same dropout_param to each dropout layer.
    self.dropout_param = {}
    if self.use_dropout:
      self.dropout_param = {'mode': 'train', 'p': dropout}
      if seed is not None:
        self.dropout_param['seed'] = seed

    # With batch normalization we need to keep track of running means and
    # variances, so we need to pass a special bn_param object to each batch
    # normalization layer. You should pass self.bn_params[0] to the forward pass
    # of the first batch normalization layer, self.bn_params[1] to the forward
    # pass of the second batch normalization layer, etc.
    self.bn_params = []
    if self.use_batchnorm:
      self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers -␣
↪1)]

    # Cast all parameters to the correct datatype
```

```
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    # ============================================================ #

    affine_caches = {}
    relu_caches = {}
    batchnorm_caches = {}
    dropout_caches = {}

    x = X
    for i in range(self.num_layers - 1):
        digit = str(i+1)

        x, affine_caches[digit] = affine_forward(x=x, w=self.params['W' +␣
↪digit], b=self.params['b' + digit])
        # x, caches[digit] = affine_relu_forward(x=x, w=self.params['W' +␣
↪digit], b=self.params['b' + digit])

        if self.use_batchnorm:
            x, batchnorm_caches[digit] = batchnorm_forward(x=x, gamma=self.
↪params['gamma' + digit], beta=self.params['beta' + digit], bn_param=self.
↪bn_params[i])
```

```
        x, relu_caches[digit] = relu_forward(x=x)

        if self.use_dropout:
            x, dropout_caches[digit] = dropout_forward(x, self.dropout_param)

    # Last layer do affine_forward
    digit = str(self.num_layers)
    scores, affine_caches[digit] = affine_forward(x=x, w=self.params['W' +␣
↪digit], b=self.params['b' + digit])

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

    # If test mode return early
    if mode == 'test':
      return scores

    loss, grads = 0.0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backwards pass of the FC net and store the gradients
    #   in the grads dict, so that grads[k] is the gradient of self.params[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    # ============================================================ #

    loss, dL = softmax_loss(scores, y)
    reg_loss_sum = 0
    for i in range(self.num_layers - 1):
        reg_loss_sum += np.sum(np.square(self.params['W' + str(i+1)]))
    loss += 0.5 * self.reg * reg_loss_sum

    # First step back do affine_backward: scores, caches[digit]
    digit = str(self.num_layers)
    dx, grads['W' + digit], grads['b' + digit] = affine_backward(dL,␣
↪affine_caches[digit])
    grads['W' + digit] += self.reg * self.params['W' + digit]

    for i in reversed(range(self.num_layers - 1)):
        digit = str(i+1)

        if self.use_dropout:
            dx = dropout_backward(dx, dropout_caches[digit])

        dx = relu_backward(dx, relu_caches[digit])
```

```
        # dx, grads['W' + digit], grads['b' + digit] = affine_relu_backward(dx,␣
↪caches[digit])

        if self.use_batchnorm:
            dx, grads['gamma' + digit], grads['beta' + digit] =␣
↪batchnorm_backward(dx, batchnorm_caches[digit])

        dx, grads['W' + digit], grads['b' + digit] = affine_backward(dx,␣
↪affine_caches[digit])

        grads['W' + digit] += self.reg * self.params['W' + digit]


    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
    return loss, grads
```