Homework 4

1 Problem 1: Optimization

1.1 Problem Statement

(35 points) Implementing different optimizers for a fully connected network. Complete the Optimization.ipynb Jupyter notebook. Print out the entire workbook and relevant code and submit it as a pdf to gradescope. Download the CIFAR-10 dataset, as you did in HW #2 and #3.

1.2 Solutions

Optimization

February 10, 2021

0.1 Optimization for Fully Connected Networks

In this notebook, we will implement different optimization rules for gradient descent. We have provided starter code; however, you will need to copy and paste your code from your implementation of the modular fully connected nets in HW #3 to build upon this.

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).

[17]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net 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))))
```

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

```
%reload_ext autoreload
```

[18]:
```
# 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,)
```

0.2 Building upon your HW #3 implementation

Copy and paste the following functions from your HW #3 implementation of a modular FC net:

- `affine_forward` in `nndl/layers.py`
- `affine_backward` in `nndl/layers.py`
- `relu_forward` in `nndl/layers.py`
- `relu_backward` in `nndl/layers.py`
- `affine_relu_forward` in `nndl/layer_utils.py`
- `affine_relu_backward` in `nndl/layer_utils.py`
- The `FullyConnectedNet` class in `nndl/fc_net.py`

0.2.1 Test all functions you copy and pasted

[19]:
```
from nndl.layer_tests import *

affine_forward_test(); print('\n')
affine_backward_test(); print('\n')
relu_forward_test(); print('\n')
relu_backward_test(); print('\n')
affine_relu_test(); print('\n')
fc_net_test()
```

```
If affine_forward function is working, difference should be less than 1e-9:
difference: 9.769849468192957e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 3.3294055798277563e-10
dw error: 4.49861745293341e-11
db error: 1.554638452499184e-11


If relu_forward function is working, difference should be around 1e-8:
```

```
difference: 4.999999798022158e-08


If relu_forward function is working, error should be less than 1e-9:
dx error: 3.275605388806931e-12


If affine_relu_forward and affine_relu_backward are working, error should be
less than 1e-9::
dx error: 1.1220504182289867e-09
dw error: 3.4433399428004154e-10
db error: 1.825267882945388e-11


Running check with reg = 0
Initial loss: 2.302843502965483
W1 relative error: 9.758605929470631e-07
W2 relative error: 2.6507832027517302e-06
W3 relative error: 6.841637118247668e-07
b1 relative error: 8.419269908098567e-09
b2 relative error: 3.34327626837078e-09
b3 relative error: 1.4827426828919928e-10
Running check with reg = 3.14
Initial loss: 6.965422476837599
W1 relative error: 1.974082043547908e-08
W2 relative error: 1.1390219864283494e-07
W3 relative error: 4.455362115056914e-07
b1 relative error: 3.1686105697505096e-08
b2 relative error: 2.562122466017697e-09
b3 relative error: 2.4382022252282145e-10
```

1 Training a larger model

In general, proceeding with vanilla stochastic gradient descent to optimize models may be fraught with problems and limitations, as discussed in class. Thus, we implement optimizers that improve on SGD.

1.1 SGD + momentum

In the following section, implement SGD with momentum. Read the `nndl/optim.py` API, which is provided by CS231n, and be sure you understand it. After, implement `sgd_momentum` in `nndl/optim.py`. Test your implementation of `sgd_momentum` by running the cell below.

[20]:
```
from nndl.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
```

```
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

print('next_w error: {}'.format(rel_error(next_w, expected_next_w)))
print('velocity error: {}'.format(rel_error(expected_velocity,␣
 ↪config['velocity'])))
```

```
next_w error: 8.882347033505819e-09
velocity error: 4.269287743278663e-09
```

1.2 SGD + Nesterov momentum

Implement `sgd_nesterov_momentum` in `ndl/optim.py`.

[21]:
```
from nndl.optim import sgd_nesterov_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_nesterov_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [0.08714,      0.15246105,  0.21778211,  0.28310316,  0.34842421],
  [0.41374526,   0.47906632,  0.54438737,  0.60970842,  0.67502947],
  [0.74035053,   0.80567158,  0.87099263,  0.93631368,  1.00163474],
  [1.06695579,   1.13227684,  1.19759789,  1.26291895,  1.32824   ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579,  0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])
```

```
print('next_w error: {}'.format(rel_error(next_w, expected_next_w)))
print('velocity error: {}'.format(rel_error(expected_velocity,␣
 ↪config['velocity'])))
```

```
next_w error: 1.0875186845081027e-08
velocity error: 4.269287743278663e-09
```

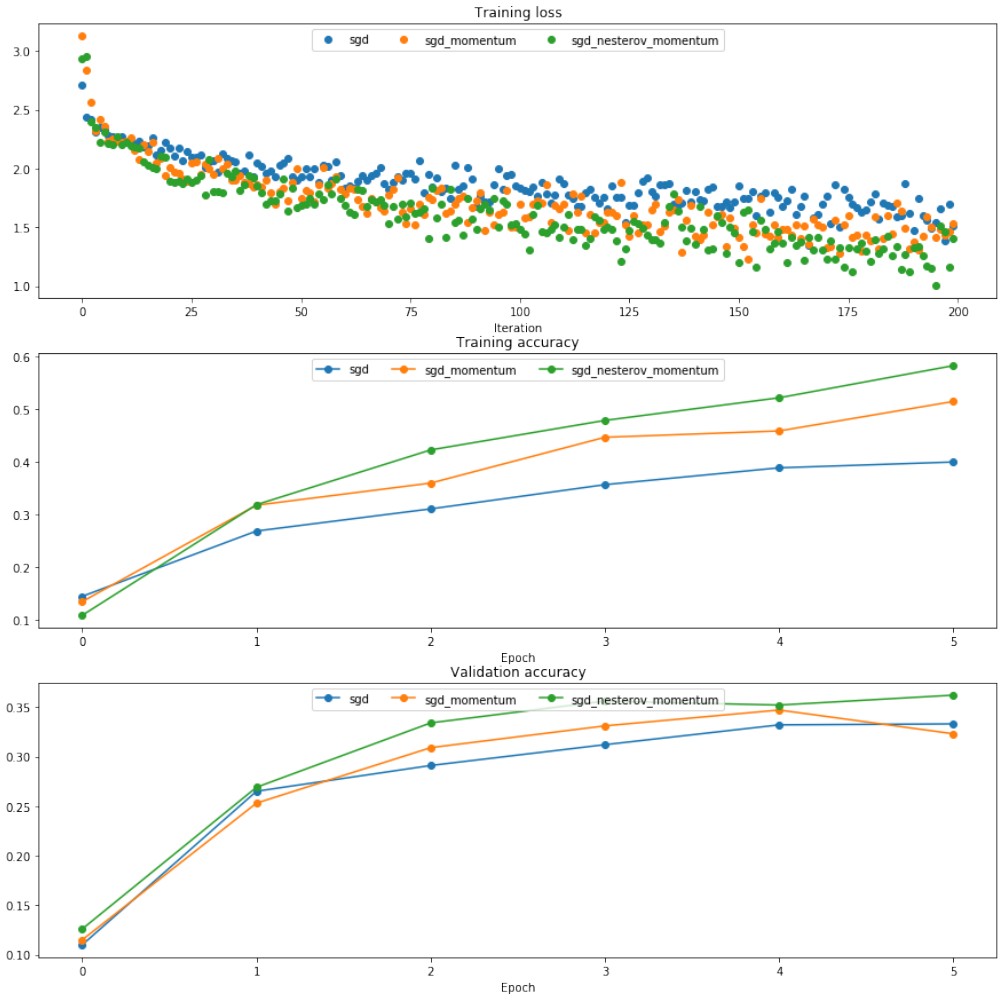
1.3 Evaluating SGD, SGD+Momentum, and SGD+NesterovMomentum

Run the following cell to train a 6 layer FC net with SGD, SGD+momentum, and SGD+Nesterov momentum. You should see that SGD+momentum achieves a better loss than SGD, and that SGD+Nesterov momentum achieves a slightly better loss (and training accuracy) than SGD+momentum.

[22]:
```
num_train = 4000
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 = {}

for update_rule in ['sgd', 'sgd_momentum', 'sgd_nesterov_momentum']:
  print('Optimizing with {}'.format(update_rule))
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=False)
  solvers[update_rule] = solver
  solver.train()
  print

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')
```

```
plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in solvers.items():
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
Optimizing with sgd
Optimizing with sgd_momentum
Optimizing with sgd_nesterov_momentum
```



1.4 RMSProp

Now we go to techniques that adapt the gradient. Implement `rmsprop` in `nndl/optim.py`. Test your implementation by running the cell below.

[23]:
```
from nndl.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
a = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'a': a}
```

```
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

print('next_w error: {}'.format(rel_error(expected_next_w, next_w)))
print('cache error: {}'.format(rel_error(expected_cache, config['a'])))
```

```
next_w error: 9.524687511038133e-08
cache error: 2.6477955807156126e-09
```

1.5 Adaptive moments

Now, implement `adam` in `nndl/optim.py`. Test your implementation by running the cell below.

[24]:
```
# Test Adam implementation; you should see errors around 1e-7 or less
from nndl.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
a = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'v': v, 'a': a, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_a = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_v = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
```

```
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

print('next_w error: {}'.format(rel_error(expected_next_w, next_w)))
print('a error: {}'.format(rel_error(expected_a, config['a'])))
print('v error: {}'.format(rel_error(expected_v, config['v'])))
```

```
next_w error: 1.1395691798535431e-07
a error: 4.208314038113071e-09
v error: 4.214963193114416e-09
```

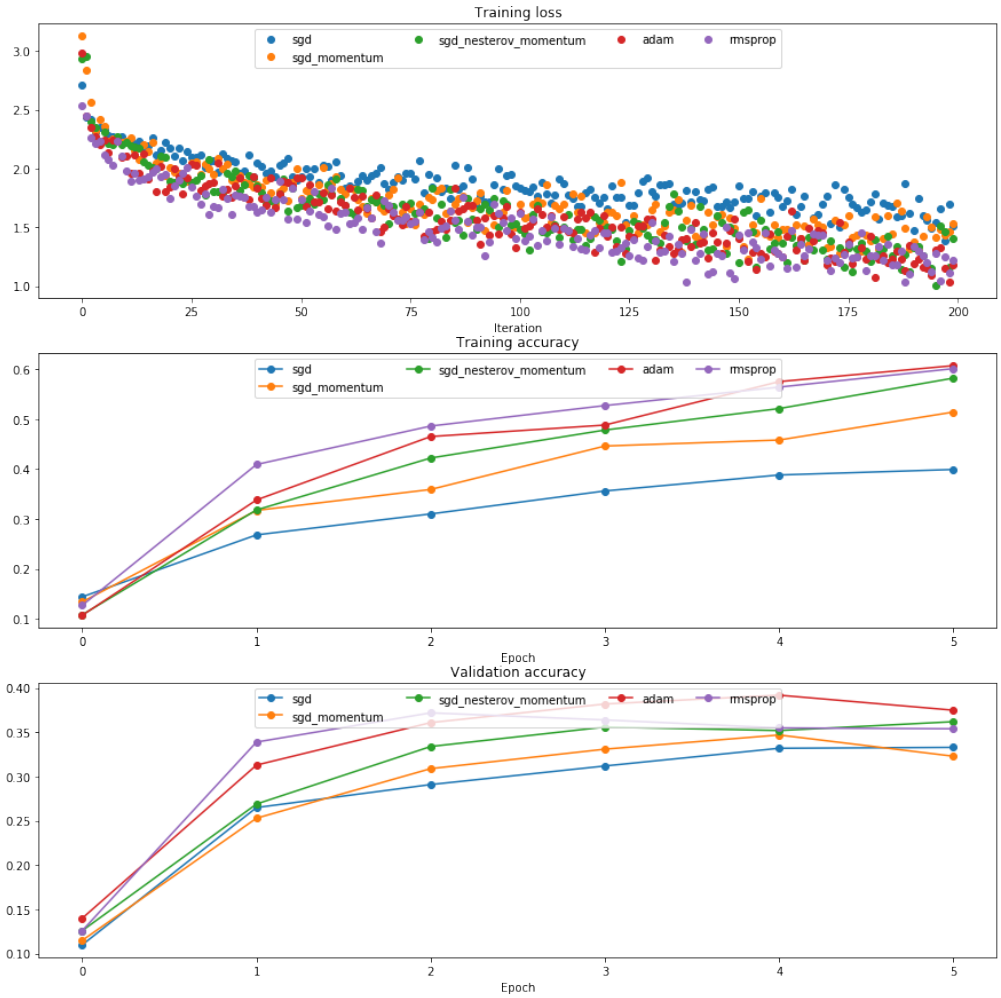
1.6 Comparing SGD, SGD+NesterovMomentum, RMSProp, and Adam

The following code will compare optimization with SGD, Momentum, Nesterov Momentum, RMSProp and Adam. In our code, we find that RMSProp, Adam, and SGD + Nesterov Momentum achieve approximately the same training error after a few training epochs.

[25]:
```
learning_rates = {'rmsprop': 2e-4, 'adam': 1e-3}

for update_rule in ['adam', 'rmsprop']:
  print('Optimizing with {}'.format(update_rule))
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=False)
  solvers[update_rule] = solver
  solver.train()
  print

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in solvers.items():
  plt.subplot(3, 1, 1)
```

```
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
Optimizing with adam
Optimizing with rmsprop
```



1.7 Easier optimization

In the following cell, we'll train a 4 layer neural network having 500 units in each hidden layer with the different optimizers, and find that it is far easier to get up to 50+% performance on CIFAR-10. After we implement batchnorm and dropout, we'll ask you to get 55+% on CIFAR-10.

[26]:
```
optimizer = 'adam'
best_model = None

layer_dims = [500, 500, 500]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale,
                          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=50)
solver.train()
```

```
(Iteration 1 / 4900) loss: 2.306725
(Epoch 0 / 10) train acc: 0.121000; val_acc: 0.148000
(Iteration 51 / 4900) loss: 1.801748
(Iteration 101 / 4900) loss: 1.589189
(Iteration 151 / 4900) loss: 1.939931
(Iteration 201 / 4900) loss: 1.679297
(Iteration 251 / 4900) loss: 1.735843
(Iteration 301 / 4900) loss: 1.549800
(Iteration 351 / 4900) loss: 1.739184
(Iteration 401 / 4900) loss: 1.721611
(Iteration 451 / 4900) loss: 1.674744
(Epoch 1 / 10) train acc: 0.451000; val_acc: 0.441000
(Iteration 501 / 4900) loss: 1.468856
(Iteration 551 / 4900) loss: 1.671042
(Iteration 601 / 4900) loss: 1.602160
(Iteration 651 / 4900) loss: 1.409814
(Iteration 701 / 4900) loss: 1.706334
(Iteration 751 / 4900) loss: 1.312938
```

```
(Iteration 801 / 4900) loss: 1.474955
(Iteration 851 / 4900) loss: 1.362951
(Iteration 901 / 4900) loss: 1.371373
(Iteration 951 / 4900) loss: 1.395546
(Epoch 2 / 10) train acc: 0.510000; val_acc: 0.444000
(Iteration 1001 / 4900) loss: 1.483582
(Iteration 1051 / 4900) loss: 1.224037
(Iteration 1101 / 4900) loss: 1.312458
(Iteration 1151 / 4900) loss: 1.469725
(Iteration 1201 / 4900) loss: 1.530206
(Iteration 1251 / 4900) loss: 1.306399
(Iteration 1301 / 4900) loss: 1.478867
(Iteration 1351 / 4900) loss: 1.231015
(Iteration 1401 / 4900) loss: 1.299022
(Iteration 1451 / 4900) loss: 1.444765
(Epoch 3 / 10) train acc: 0.522000; val_acc: 0.505000
(Iteration 1501 / 4900) loss: 1.384641
(Iteration 1551 / 4900) loss: 1.352226
(Iteration 1601 / 4900) loss: 1.443149
(Iteration 1651 / 4900) loss: 1.272070
(Iteration 1701 / 4900) loss: 1.430958
(Iteration 1751 / 4900) loss: 1.132169
(Iteration 1801 / 4900) loss: 1.271073
(Iteration 1851 / 4900) loss: 1.333373
(Iteration 1901 / 4900) loss: 1.235261
(Iteration 1951 / 4900) loss: 1.395004
(Epoch 4 / 10) train acc: 0.544000; val_acc: 0.509000
(Iteration 2001 / 4900) loss: 1.216636
(Iteration 2051 / 4900) loss: 1.157197
(Iteration 2101 / 4900) loss: 1.134426
(Iteration 2151 / 4900) loss: 1.318604
(Iteration 2201 / 4900) loss: 1.136871
(Iteration 2251 / 4900) loss: 1.413491
(Iteration 2301 / 4900) loss: 1.350720
(Iteration 2351 / 4900) loss: 1.067860
(Iteration 2401 / 4900) loss: 1.343558
(Epoch 5 / 10) train acc: 0.551000; val_acc: 0.530000
(Iteration 2451 / 4900) loss: 1.205991
(Iteration 2501 / 4900) loss: 0.909917
(Iteration 2551 / 4900) loss: 1.196699
(Iteration 2601 / 4900) loss: 1.357115
(Iteration 2651 / 4900) loss: 1.258780
(Iteration 2701 / 4900) loss: 1.045392
(Iteration 2751 / 4900) loss: 1.398324
(Iteration 2801 / 4900) loss: 1.296340
(Iteration 2851 / 4900) loss: 1.234940
(Iteration 2901 / 4900) loss: 1.257974
(Epoch 6 / 10) train acc: 0.606000; val_acc: 0.524000
```

```
(Iteration 2951 / 4900) loss: 1.079811
(Iteration 3001 / 4900) loss: 1.115626
(Iteration 3051 / 4900) loss: 1.222847
(Iteration 3101 / 4900) loss: 1.336243
(Iteration 3151 / 4900) loss: 1.235378
(Iteration 3201 / 4900) loss: 1.270102
(Iteration 3251 / 4900) loss: 0.994448
(Iteration 3301 / 4900) loss: 1.284518
(Iteration 3351 / 4900) loss: 1.109864
(Iteration 3401 / 4900) loss: 1.160492
(Epoch 7 / 10) train acc: 0.638000; val_acc: 0.517000
(Iteration 3451 / 4900) loss: 1.111910
(Iteration 3501 / 4900) loss: 1.108662
(Iteration 3551 / 4900) loss: 1.170207
(Iteration 3601 / 4900) loss: 1.008984
(Iteration 3651 / 4900) loss: 1.230990
(Iteration 3701 / 4900) loss: 0.989104
(Iteration 3751 / 4900) loss: 1.075819
(Iteration 3801 / 4900) loss: 1.039737
(Iteration 3851 / 4900) loss: 0.900732
(Iteration 3901 / 4900) loss: 0.984025
(Epoch 8 / 10) train acc: 0.647000; val_acc: 0.534000
(Iteration 3951 / 4900) loss: 1.085295
(Iteration 4001 / 4900) loss: 0.951691
(Iteration 4051 / 4900) loss: 0.861846
(Iteration 4101 / 4900) loss: 0.890610
(Iteration 4151 / 4900) loss: 0.875024
(Iteration 4201 / 4900) loss: 0.933870
(Iteration 4251 / 4900) loss: 0.922426
(Iteration 4301 / 4900) loss: 0.998661
(Iteration 4351 / 4900) loss: 0.914697
(Iteration 4401 / 4900) loss: 0.780546
(Epoch 9 / 10) train acc: 0.674000; val_acc: 0.548000
(Iteration 4451 / 4900) loss: 0.956199
(Iteration 4501 / 4900) loss: 1.082839
(Iteration 4551 / 4900) loss: 0.817555
(Iteration 4601 / 4900) loss: 0.836931
(Iteration 4651 / 4900) loss: 0.954816
(Iteration 4701 / 4900) loss: 0.924653
(Iteration 4751 / 4900) loss: 0.865339
(Iteration 4801 / 4900) loss: 0.961394
(Iteration 4851 / 4900) loss: 0.874492
(Epoch 10 / 10) train acc: 0.714000; val_acc: 0.549000
```

[27]:
```
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'])))
```

```
Validation set accuracy: 0.549
Test set accuracy: 0.529
```

2 Problem 2: Batch Normalization

2.1 Problem Statement

(35 points) Implementing batch normalization for a fully connected network. Complete the Batch-Normalization.ipynb Jupyter notebook. Print out the entire workbook and relevant code and submit it as a pdf to gradescope.

2.2 Solutions

Batch-Normalization

February 10, 2021

1 Batch Normalization

In this notebook, you will implement the batch normalization layers of a neural network to increase its performance. Please review the details of batch normalization from the lecture notes.

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).

[39]:
```
## 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))))
```

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

```
%reload_ext autoreload
```

[40]:
```
# 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 Batchnorm forward pass

Implement the training time batchnorm forward pass, `batchnorm_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

[41]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before batch normalization:')
print('  means: ', a.mean(axis=0))
print('  stds: ', a.std(axis=0))

# Means should be close to zero and stds close to one
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = batchnorm_forward(a, np.ones(D3), np.zeros(D3), {'mode': 'train'})
print('  mean: ', a_norm.mean(axis=0))
print('  std: ', a_norm.std(axis=0))

# Now means should be close to beta and stds close to gamma
gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print('After batch normalization (nontrivial gamma, beta)')
print('  means: ', a_norm.mean(axis=0))
print('  stds: ', a_norm.std(axis=0))
```

```
Before batch normalization:
  means:  [ 29.7799643  -20.32858792 -10.63100559]
  stds:  [35.52222372 31.93401087 31.51214995]
After batch normalization (gamma=1, beta=0)
  mean:  [-4.17443857e-16  3.18634008e-16 -3.33066907e-18]
  std:  [1.         1.         0.99999999]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [1.         1.99999999 2.99999998]
```

Implement the testing time batchnorm forward pass, `batchnorm_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

[42]:
```
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.

N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)
for t in np.arange(50):
  X = np.random.randn(N, D1)
  a = np.maximum(0, X.dot(W1)).dot(W2)
  batchnorm_forward(a, gamma, beta, bn_param)
bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=0))
print('  stds: ', a_norm.std(axis=0))
```

```
After batch normalization (test-time):
  means:  [-0.03579019  0.07465793  0.12493896]
  stds:  [1.019034   1.09216426 1.09617924]
```

1.2 Batchnorm backward pass

Implement the backward pass for the batchnorm layer, `batchnorm_backward` in `nndl/layers.py`. Check your implementation by running the following cell.

[43]:
```
# Gradient check batchnorm backward pass

N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

bn_param = {'mode': 'train'}
fx = lambda x: batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.3129050966552026e-08
dgamma error:  8.78421050457484e-11
dbeta error:  6.6491911418713455e-12
```

1.3 Implement a fully connected neural network with batchnorm layers

Modify the `FullyConnectedNet()` class in `nndl/fc_net.py` to incorporate batchnorm layers. You will need to modify the class in the following areas:

(1) The gammas and betas need to be initialized to 1's and 0's respectively in `__init__`.

(2) The `batchnorm_forward` layer needs to be inserted between each affine and relu layer (except in the output layer) in a forward pass computation in `loss`. You may find it helpful to write an `affine_batchnorm_relu()` layer in `nndl/layer_utils.py` although this is not necessary.

(3) The `batchnorm_backward` layer has to be appropriately inserted when calculating gradients.

After you have done the appropriate modifications, check your implementation by running the following cell.

Note, while the relative error for W3 should be small, as we backprop gradients more, you may find the relative error increases. Our relative error for W1 is on the order of 1e-4.

[46]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))
```

```
for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64,
                            use_batchnorm=True)

  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])))
  if reg == 0: print('\n')
```

```
Running check with reg =  0
Initial loss:  2.4477680636130836
W1 relative error: 5.4953082949795227e-05
W2 relative error: 3.589167845868008e-05
W3 relative error: 4.034792680988542e-10
b1 relative error: 0.002220457151480559
b2 relative error: 0.0022204460492503126
b3 relative error: 1.122114974113118e-10
beta1 relative error: 6.55189827159565e-09
beta2 relative error: 4.518131245421938e-09
gamma1 relative error: 6.574892843351459e-09
gamma2 relative error: 8.427052451317531e-09


Running check with reg =  3.14
Initial loss:  6.875099213505022
W1 relative error: 9.425837715597008e-06
W2 relative error: 3.148443999144469e-06
W3 relative error: 3.335792306257643e-08
b1 relative error: 2.220446049250313e-08
b2 relative error: 1.1102230246251565e-08
b3 relative error: 1.8342710670657887e-10
beta1 relative error: 1.2420664244591935e-08
beta2 relative error: 6.339878500295625e-08
gamma1 relative error: 1.2387325283193854e-08
gamma2 relative error: 9.296489046249204e-08
```

1.4 Training a deep fully connected network with batch normalization.

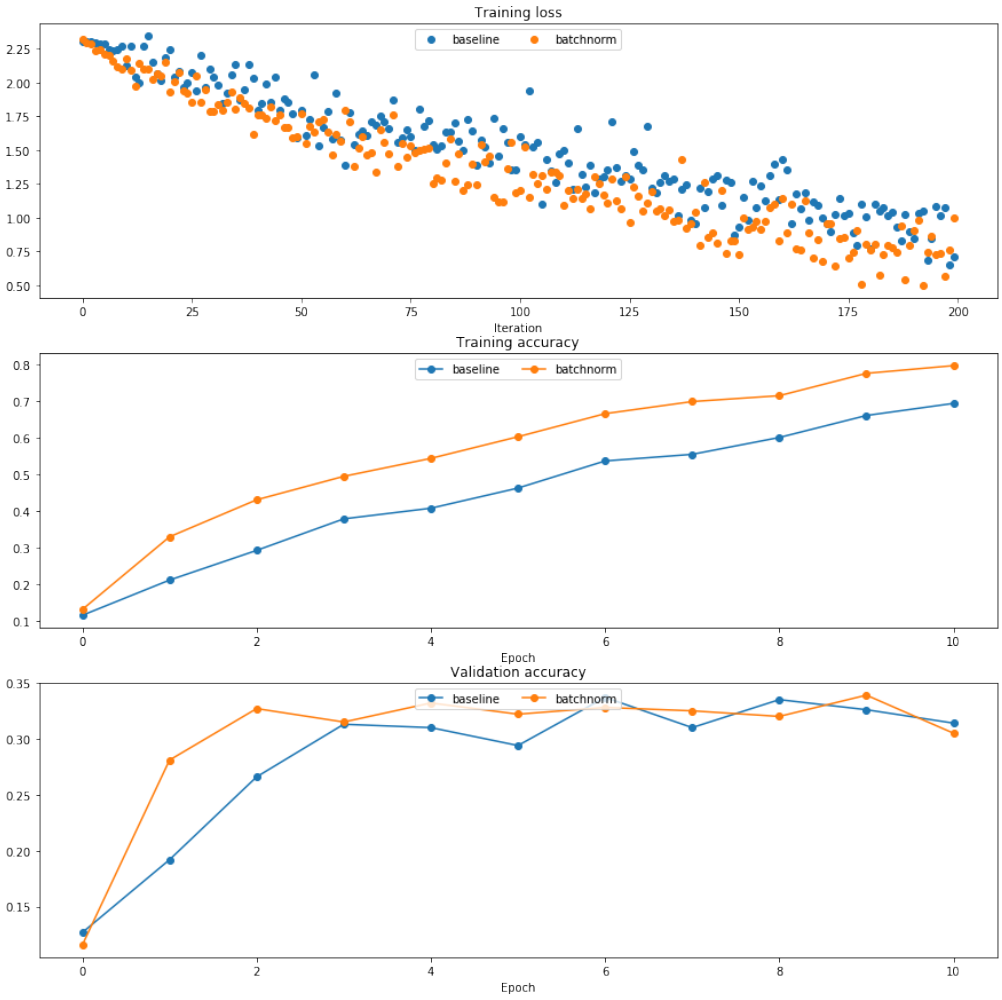
To see if batchnorm helps, let's train a deep neural network with and without batch normalization.

[47]:
```
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
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'],
}

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪use_batchnorm=True)
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪use_batchnorm=False)

bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
bn_solver.train()

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
solver.train()
```

```
(Iteration 1 / 200) loss: 2.315853
(Epoch 0 / 10) train acc: 0.131000; val_acc: 0.116000
(Epoch 1 / 10) train acc: 0.329000; val_acc: 0.281000
(Epoch 2 / 10) train acc: 0.430000; val_acc: 0.327000
(Epoch 3 / 10) train acc: 0.494000; val_acc: 0.315000
(Epoch 4 / 10) train acc: 0.543000; val_acc: 0.332000
(Epoch 5 / 10) train acc: 0.602000; val_acc: 0.322000
(Epoch 6 / 10) train acc: 0.665000; val_acc: 0.328000
(Epoch 7 / 10) train acc: 0.698000; val_acc: 0.325000
(Epoch 8 / 10) train acc: 0.714000; val_acc: 0.320000
(Epoch 9 / 10) train acc: 0.775000; val_acc: 0.339000
(Epoch 10 / 10) train acc: 0.796000; val_acc: 0.305000
(Iteration 1 / 200) loss: 2.302447
```

```
(Epoch 0 / 10) train acc: 0.115000; val_acc: 0.127000
(Epoch 1 / 10) train acc: 0.211000; val_acc: 0.192000
(Epoch 2 / 10) train acc: 0.292000; val_acc: 0.266000
(Epoch 3 / 10) train acc: 0.378000; val_acc: 0.313000
(Epoch 4 / 10) train acc: 0.407000; val_acc: 0.310000
(Epoch 5 / 10) train acc: 0.462000; val_acc: 0.294000
(Epoch 6 / 10) train acc: 0.536000; val_acc: 0.337000
(Epoch 7 / 10) train acc: 0.554000; val_acc: 0.310000
(Epoch 8 / 10) train acc: 0.600000; val_acc: 0.335000
(Epoch 9 / 10) train acc: 0.660000; val_acc: 0.326000
(Epoch 10 / 10) train acc: 0.693000; val_acc: 0.314000
```

[48]:
```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(solver.loss_history, 'o', label='baseline')
plt.plot(bn_solver.loss_history, 'o', label='batchnorm')

plt.subplot(3, 1, 2)
plt.plot(solver.train_acc_history, '-o', label='baseline')
plt.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

plt.subplot(3, 1, 3)
plt.plot(solver.val_acc_history, '-o', label='baseline')
plt.plot(bn_solver.val_acc_history, '-o', label='batchnorm')

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```



1.5 Batchnorm and initialization

The following cells run an experiment where for a deep network, the initialization is varied. We do training for when batchnorm layers are and are not included.

[49]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
```

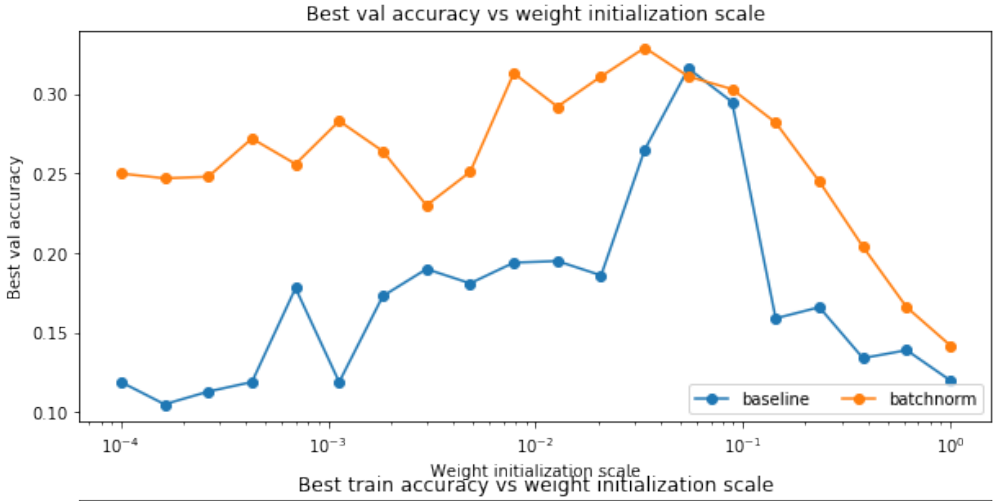
```
  'y_val': data['y_val'],
}

bn_solvers = {}
solvers = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale {} / {}'.format(i + 1, len(weight_scales)))
  bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
↪use_batchnorm=True)
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
↪use_batchnorm=False)

  bn_solver = Solver(bn_model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
  bn_solver.train()
  bn_solvers[weight_scale] = bn_solver

  solver = Solver(model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
  solver.train()
  solvers[weight_scale] = solver
```

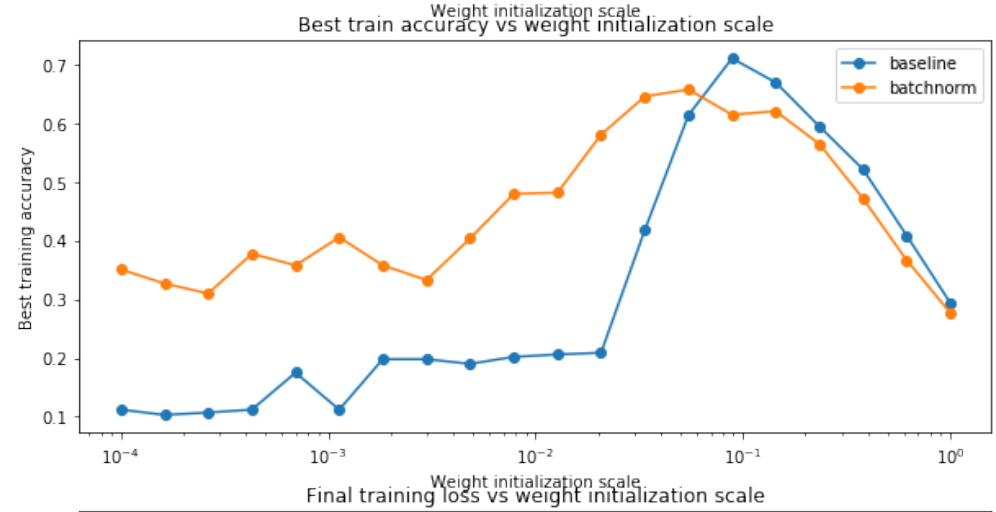
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
```

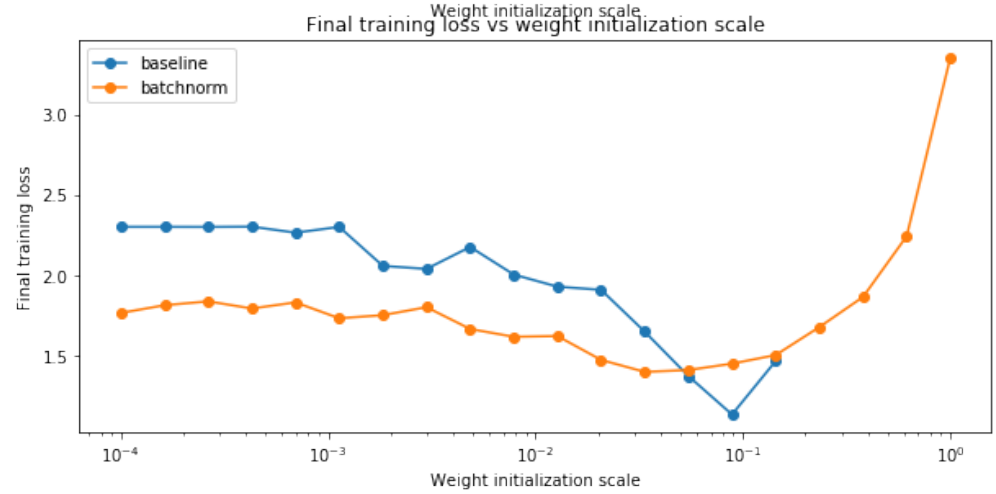
```
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

[50]:
```
# Plot results of weight scale experiment
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

for ws in weight_scales:
  best_train_accs.append(max(solvers[ws].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers[ws].train_acc_history))

  best_val_accs.append(max(solvers[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers[ws].loss_history[-100:]))

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()

plt.gcf().set_size_inches(10, 15)
```

```
plt.show()
```


Best val accuracy vs weight initialization scale
Best val accuracy
0.30
0.25
0.20
0.15
0.10
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale


Best train accuracy vs weight initialization scale
Best training accuracy
0.7
0.6
0.5
0.4
0.3
0.2
0.1
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale


Final training loss vs weight initialization scale
Final training loss
3.0
2.5
2.0
1.5
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale

1.6 Question:

In the cell below, summarize the findings of this experiment, and WHY these results make sense.

1.7 Answer:

Using batchnorm reduces the strong dependence on initialization. In the validation and training accuracy one can see the accuracy when using batchnorm with different weight initializations varies less than the baseline. The training loss when using batchnorm is also relatively the same for weight initialization scales under 10^(-1)

3 Problem 3: Dropout

3.1 Problem Statement

(30 points) Implementing dropout for a fully connected network, and optimizing it. Complete the Dropout.ipynb Jupyter notebook. Print out the entire workbook and relevant code and submit it as a pdf to gradescope.

3.2 Solutions

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).

[2]:
```
## 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))))
```

[3]:
```
# 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.

[4]:
```
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:  9.998250424376245
Mean of train-time output:  9.986928712252833
Mean of test-time output:  9.998250424376245
Fraction of train-time output set to zero:  0.700424
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.998250424376245
Mean of train-time output:  9.984458472057762
Mean of test-time output:  9.998250424376245
Fraction of train-time output set to zero:  0.400796
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.998250424376245
Mean of train-time output:  10.012624110628005
Mean of test-time output:  9.998250424376245
Fraction of train-time output set to zero:  0.248868
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:

[5]:
```
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:  5.445609591683889e-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).

[6]:
```
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: 1.4026015558098908e-10


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: 8.009887154529434e-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.

[7]:
```
# 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'],
```

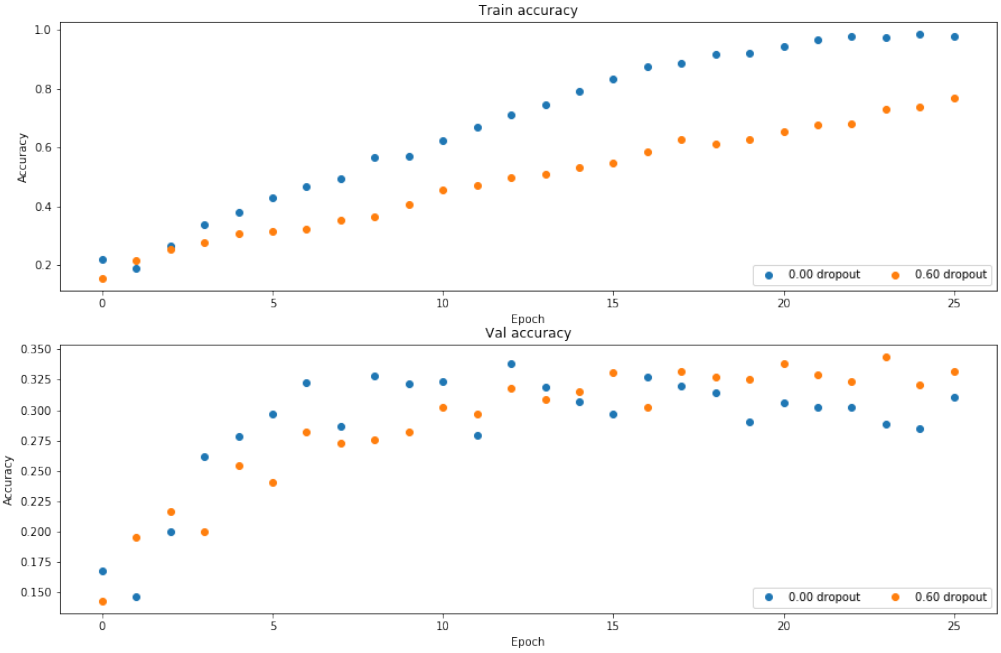
```
}

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: 1.299273
(Epoch 21 / 25) train acc: 0.676000; val_acc: 0.329000
(Epoch 22 / 25) train acc: 0.682000; val_acc: 0.324000
(Epoch 23 / 25) train acc: 0.730000; val_acc: 0.344000
(Epoch 24 / 25) train acc: 0.740000; val_acc: 0.321000
(Epoch 25 / 25) train acc: 0.770000; val_acc: 0.332000
```

[8]:
```
# 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 although the training accuracy is not as good as with 0 dropout, the validation accuracy is slightly better. Without dropout, the model is 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.

[19]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #

decay = 0.9
dropout = 0.65
weight_scale = 0.01
model = FullyConnectedNet([500, 1000, 500], weight_scale=weight_scale,␣
 ↪dropout=dropout,
                          use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=30, batch_size= 500,
                update_rule='adam',
                optim_config={
                'learning_rate': 9e-4,
                },
                lr_decay = decay,
                verbose=True, print_every=100)
solver.train()

best_model = solver.model

y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_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 / 2940) loss: 2.328987
(Epoch 0 / 30) train acc: 0.261000; val_acc: 0.246000
(Epoch 1 / 30) train acc: 0.482000; val_acc: 0.457000
(Iteration 101 / 2940) loss: 1.522141
(Epoch 2 / 30) train acc: 0.547000; val_acc: 0.504000
(Iteration 201 / 2940) loss: 1.426827
(Epoch 3 / 30) train acc: 0.551000; val_acc: 0.532000
(Iteration 301 / 2940) loss: 1.416075
(Epoch 4 / 30) train acc: 0.543000; val_acc: 0.540000
(Iteration 401 / 2940) loss: 1.364265
```

```
(Epoch 5 / 30) train acc: 0.563000; val_acc: 0.534000
(Iteration 501 / 2940) loss: 1.287728
(Epoch 6 / 30) train acc: 0.603000; val_acc: 0.558000
(Iteration 601 / 2940) loss: 1.224824
(Epoch 7 / 30) train acc: 0.625000; val_acc: 0.564000
(Iteration 701 / 2940) loss: 1.123295
(Epoch 8 / 30) train acc: 0.601000; val_acc: 0.557000
(Iteration 801 / 2940) loss: 1.074512
(Epoch 9 / 30) train acc: 0.633000; val_acc: 0.569000
(Iteration 901 / 2940) loss: 1.128574
(Epoch 10 / 30) train acc: 0.646000; val_acc: 0.559000
(Iteration 1001 / 2940) loss: 1.087263
(Epoch 11 / 30) train acc: 0.687000; val_acc: 0.563000
(Iteration 1101 / 2940) loss: 1.060235
(Epoch 12 / 30) train acc: 0.679000; val_acc: 0.570000
(Iteration 1201 / 2940) loss: 1.035633
(Epoch 13 / 30) train acc: 0.716000; val_acc: 0.583000
(Iteration 1301 / 2940) loss: 0.988168
(Epoch 14 / 30) train acc: 0.721000; val_acc: 0.581000
(Iteration 1401 / 2940) loss: 0.961157
(Epoch 15 / 30) train acc: 0.712000; val_acc: 0.578000
(Iteration 1501 / 2940) loss: 0.987495
(Epoch 16 / 30) train acc: 0.708000; val_acc: 0.578000
(Iteration 1601 / 2940) loss: 1.002675
(Epoch 17 / 30) train acc: 0.731000; val_acc: 0.599000
(Iteration 1701 / 2940) loss: 0.913845
(Epoch 18 / 30) train acc: 0.732000; val_acc: 0.578000
(Iteration 1801 / 2940) loss: 0.898629
(Epoch 19 / 30) train acc: 0.741000; val_acc: 0.587000
(Iteration 1901 / 2940) loss: 0.876350
(Epoch 20 / 30) train acc: 0.740000; val_acc: 0.587000
(Iteration 2001 / 2940) loss: 0.925863
(Epoch 21 / 30) train acc: 0.740000; val_acc: 0.591000
(Iteration 2101 / 2940) loss: 0.777185
(Epoch 22 / 30) train acc: 0.752000; val_acc: 0.603000
(Iteration 2201 / 2940) loss: 0.844658
(Epoch 23 / 30) train acc: 0.760000; val_acc: 0.597000
(Iteration 2301 / 2940) loss: 0.884060
(Epoch 24 / 30) train acc: 0.749000; val_acc: 0.596000
(Iteration 2401 / 2940) loss: 0.864935
(Epoch 25 / 30) train acc: 0.784000; val_acc: 0.596000
(Iteration 2501 / 2940) loss: 0.830847
(Epoch 26 / 30) train acc: 0.766000; val_acc: 0.582000
(Iteration 2601 / 2940) loss: 0.788061
(Epoch 27 / 30) train acc: 0.781000; val_acc: 0.597000
(Iteration 2701 / 2940) loss: 0.784128
(Epoch 28 / 30) train acc: 0.792000; val_acc: 0.604000
(Iteration 2801 / 2940) loss: 0.824073
```

```
(Epoch 29 / 30) train acc: 0.779000; val_acc: 0.599000
(Iteration 2901 / 2940) loss: 0.754507
(Epoch 30 / 30) train acc: 0.793000; val_acc: 0.596000
Validation set accuracy: 0.607
Test set accuracy: 0.602
```

[ ]:

A Helper Functions

A.1 optim.py

```
1  import numpy as np
2
3  """
4  This code was originally written for CS 231n at Stanford University
5  (cs231n.stanford.edu).  It has been modified in various areas for use ...
       in the
6  ECE 239AS class at UCLA.  This includes the descriptions of what code to
7  implement as well as some slight potential changes in variable names to be
8  consistent with class nomenclature.  We thank Justin Johnson & Serena ...
       Yeung for
9  permission to use this code.  To see the original version, please visit
10 cs231n.stanford.edu.
11 """
12
13 """
14 This file implements various first-order update rules that are commonly ...
       used for
15 training neural networks. Each update rule accepts current weights and the
16 gradient of the loss with respect to those weights and produces the ...
       next set of
17 weights. Each update rule has the same interface:
18
19 def update(w, dw, config=None):
20
21 Inputs:
22   - w: A numpy array giving the current weights.
23   - dw: A numpy array of the same shape as w giving the gradient of the
24     loss with respect to w.
```

```
25    - config: A dictionary containing hyperparameter values such as ...
          learning rate,
26      momentum, etc. If the update rule requires caching values over many
27      iterations, then config will also hold these cached values.
28
29  Returns:
30    - next_w: The next point after the update.
31    - config: The config dictionary to be passed to the next iteration of the
32      update rule.
33
34  NOTE: For most update rules, the default learning rate will probably ...
        not perform
35  well; however the default values of the other hyperparameters should ...
        work well
36  for a variety of different problems.
37
38  For efficiency, update rules may perform in-place updates, mutating w and
39  setting next_w equal to w.
40  """
41
42
43  def sgd(w, dw, config=None):
44    """
45    Performs vanilla stochastic gradient descent.
46
47    config format:
48    - learning_rate: Scalar learning rate.
49    """
50    if config is None: config = {}
51    config.setdefault('learning_rate', 1e-2)
52
53    w -= config['learning_rate'] * dw
54    return w, config
```

```
55
56
57 def sgd_momentum(w, dw, config=None):
58   """
59   Performs stochastic gradient descent with momentum.
60
61   config format:
62   - learning_rate: Scalar learning rate.
63   - momentum: Scalar between 0 and 1 giving the momentum value.
64     Setting momentum = 0 reduces to sgd.
65   - velocity: A numpy array of the same shape as w and dw used to store ...
       a moving
66     average of the gradients.
67   """
68   if config is None: config = {}
69   config.setdefault('learning_rate', 1e-2)
70   config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't ...
       there
71   v = config.get('velocity', np.zeros_like(w))   # gets velocity, else ...
       sets it to zero.
72
73   # ================================================================ #
74   # YOUR CODE HERE:
75   #   Implement the momentum update formula.  Return the updated weights
76   #   as next_w, and the updated velocity as v.
77   # ================================================================ #
78
79   alpha = config['momentum']
80   v = alpha*v - config['learning_rate']*dw
81   next_w = w + v
82
83   # ================================================================ #
84   # END YOUR CODE HERE
```

```
85    # ================================================================ #
86
87    config['velocity'] = v
88
89    return next_w, config
90
91  def sgd_nesterov_momentum(w, dw, config=None):
92    """
93    Performs stochastic gradient descent with Nesterov momentum.
94
95    config format:
96    - learning_rate: Scalar learning rate.
97    - momentum: Scalar between 0 and 1 giving the momentum value.
98      Setting momentum = 0 reduces to sgd.
99    - velocity: A numpy array of the same shape as w and dw used to store ...
          a moving
100     average of the gradients.
101   """
102   if config is None: config = {}
103   config.setdefault('learning_rate', 1e-2)
104   config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't ...
          there
105   v = config.get('velocity', np.zeros_like(w))   # gets velocity, else ...
          sets it to zero.
106
107   # ================================================================ #
108   # YOUR CODE HERE:
109   #   Implement the momentum update formula.  Return the updated weights
110   #   as next_w, and the updated velocity as v.
111   # ================================================================ #
112
113   alpha = config['momentum']
114   v_old = v
```

```
115
116     # nesterov momentum calculates the gradient AFTER taking a step along ...
            direction of momentum
117     v = alpha*v - config['learning_rate']*dw
118     next_w = w + v + alpha*(v-v_old)
119
120     # ================================================================ #
121     # END YOUR CODE HERE
122     # ================================================================ #
123
124     config['velocity'] = v
125
126     return next_w, config
127
128 def rmsprop(w, dw, config=None):
129     """
130     Uses the RMSProp update rule, which uses a moving average of squared ...
            gradient
131     values to set adaptive per-parameter learning rates.
132
133     config format:
134     - learning_rate: Scalar learning rate.
135     - decay_rate: Scalar between 0 and 1 giving the decay rate for the ...
            squared
136       gradient cache.
137     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
138     - beta: Moving average of second moments of gradients.
139     """
140     if config is None: config = {}
141     config.setdefault('learning_rate', 1e-2)
142     config.setdefault('decay_rate', 0.99)
143     config.setdefault('epsilon', 1e-8)
144     config.setdefault('a', np.zeros_like(w))
```

```
145
146     next_w = None
147
148     # ================================================================ #
149     # YOUR CODE HERE:
150     #   Implement RMSProp.  Store the next value of w as next_w.  You need
151     #   to also store in config['a'] the moving average of the second
152     #   moment gradients, so they can be used for future gradients. ...
            Concretely,
153     #   config['a'] corresponds to "a" in the lecture notes.
154     # ================================================================ #
155     nu = config['epsilon']
156     beta = config['decay_rate']
157     a = config['a']
158
159     a = beta*a + (1-beta)*dw*dw
160     next_w  = w - config['learning_rate']/(np.sqrt(a)+nu)*dw
161
162     config['a'] = a
163     # ================================================================ #
164     # END YOUR CODE HERE
165     # ================================================================ #
166
167     return next_w, config
168
169
170 def adam(w, dw, config=None):
171     """
172     Uses the Adam update rule, which incorporates moving averages of both the
173     gradient and its square and a bias correction term.
174
175     config format:
176     - learning_rate: Scalar learning rate.
```

```
177     - beta1: Decay rate for moving average of first moment of gradient.
178     - beta2: Decay rate for moving average of second moment of gradient.
179     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
180     - m: Moving average of gradient.
181     - v: Moving average of squared gradient.
182     - t: Iteration number.
183     """
184     if config is None: config = {}
185     config.setdefault('learning_rate', 1e-3)
186     config.setdefault('beta1', 0.9)
187     config.setdefault('beta2', 0.999)
188     config.setdefault('epsilon', 1e-8)
189     config.setdefault('v', np.zeros_like(w))
190     config.setdefault('a', np.zeros_like(w))
191     config.setdefault('t', 0)
192
193     next_w = None
194
195     # ================================================================ #
196     # YOUR CODE HERE:
197     #   Implement Adam.  Store the next value of w as next_w.  You need
198     #   to also store in config['a'] the moving average of the second
199     #   moment gradients, and in config['v'] the moving average of the
200     #   first moments.  Finally, store in config['t'] the increasing time.
201     # ================================================================ #
202     # variable definitions
203     v = config['v']
204     a = config['a']
205     nu = config['epsilon']
206     B1 = config['beta1']
207     B2 = config['beta2']
208
209     # time update
```

```
210     config['t'] += 1
211     t = config['t']
212 
213     # first moment update
214     v = B1*v + (1- B1)*dw
215 
216     # second moment update (gradient normalization)
217     a = B2*a + (1- B2)*dw*dw
218 
219     # bias correction
220     v_tild = 1/(1-B1**t)*v
221     a_tild = 1/(1-B2**t)*a
222 
223     # gradient step
224     next_w = w - config['learning_rate']/(np.sqrt(a_tild)+nu)*v_tild
225 
226     config['a'] = a
227     config['v'] = v
228 
229     # ================================================================ #
230     # END YOUR CODE HERE
231     # ================================================================ #
232 
233     return next_w, config
```

A.2 fc_net.py

```
1  import numpy as np
2  import pdb
3
4  from .layers import *
5  from .layer_utils import *
6
7  """
8  This code was originally written for CS 231n at Stanford University
9  (cs231n.stanford.edu).  It has been modified in various areas for use ...
       in the
10 ECE 239AS class at UCLA.  This includes the descriptions of what code to
11 implement as well as some slight potential changes in variable names to be
12 consistent with class nomenclature.  We thank Justin Johnson & Serena ...
       Yeung for
13 permission to use this code.  To see the original version, please visit
14 cs231n.stanford.edu.
15 """
16
17 class TwoLayerNet(object):
18   """
19   A two-layer fully-connected neural network with ReLU nonlinearity and
20   softmax loss that uses a modular layer design. We assume an input ...
         dimension
21   of D, a hidden dimension of H, and perform classification over C classes.
22
23   The architecure should be affine - relu - affine - softmax.
24
25   Note that this class does not implement gradient descent; instead, it
26   will interact with a separate Solver object that is responsible for ...
         running
```

```
27    optimization.
28
29    The learnable parameters of the model are stored in the dictionary
30    self.params that maps parameter names to numpy arrays.
31    """
32
33    def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
34                 dropout=0, weight_scale=1e-3, reg=0.0):
35      """
36      Initialize a new network.
37
38      Inputs:
39      - input_dim: An integer giving the size of the input
40      - hidden_dims: An integer giving the size of the hidden layer
41      - num_classes: An integer giving the number of classes to classify
42      - dropout: Scalar between 0 and 1 giving dropout strength.
43      - weight_scale: Scalar giving the standard deviation for random
44        initialization of the weights.
45      - reg: Scalar giving L2 regularization strength.
46      """
47      self.params = {}
48      self.reg = reg
49
50      # ============================================================ #
51      # YOUR CODE HERE:
52      #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
53      #   self.params['W2'], self.params['b1'] and self.params['b2']. The
54      #   biases are initialized to zero and the weights are initialized
55      #   so that each parameter has mean 0 and standard deviation ...
           weight_scale.
56      #   The dimensions of W1 should be (input_dim, hidden_dim) and the
57      #   dimensions of W2 should be (hidden_dims, num_classes)
58      # ============================================================ #
```

```
59
60      self.params['W1'] = weight_scale * np.random.randn(input_dim, ...
            hidden_dims)
61      self.params['b1'] = np.zeros(hidden_dims)
62      self.params['W2'] = weight_scale * np.random.randn(hidden_dims, ...
            num_classes)
63      self.params['b2'] = np.zeros(num_classes)
64
65      # ============================================================ #
66      # END YOUR CODE HERE
67      # ============================================================ #
68
69    def loss(self, X, y=None):
70      """
71      Compute loss and gradient for a minibatch of data.
72
73      Inputs:
74      - X: Array of input data of shape (N, d_1, ..., d_k)
75      - y: Array of labels, of shape (N,). y[i] gives the label for X[i].
76
77      Returns:
78      If y is None, then run a test-time forward pass of the model and ...
            return:
79      - scores: Array of shape (N, C) giving classification scores, where
80        scores[i, c] is the classification score for X[i] and class c.
81
82      If y is not None, then run a training-time forward and backward ...
            pass and
83      return a tuple of:
84      - loss: Scalar value giving the loss
85      - grads: Dictionary with the same keys as self.params, mapping ...
            parameter
86        names to gradients of the loss with respect to those parameters.
```

```
87      """
88      scores = None
89
90      # ============================================================ #
91      # YOUR CODE HERE:
92      #   Implement the forward pass of the two-layer neural network. Store
93      #   the class scores as the variable 'scores'.  Be sure to use the ...
            layers
94      #   you prior implemented.
95      # ============================================================ #
96
97      # Unpack variables from the params dictionary
98      W1, b1 = self.params['W1'], self.params['b1']
99      W2, b2 = self.params['W2'], self.params['b2']
100
101     h, h_cache = affine_relu_forward(X,W1,b1)
102     scores, z_cache = affine_forward(h,W2,b2)
103
104     # ============================================================ #
105     # END YOUR CODE HERE
106     # ============================================================ #
107
108     # If y is None then we are in test mode so just return scores
109     if y is None:
110       return scores
111
112     loss, grads = 0, {}
113     # ============================================================ #
114     # YOUR CODE HERE:
115     #   Implement the backward pass of the two-layer neural net.  Store
116     #   the loss as the variable 'loss' and store the gradients in the
117     #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
118     #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
```

```
119       #   i.e., grads[k] holds the gradient for self.params[k].
120       #
121       #   Add L2 regularization, where there is an added cost ...
              0.5*self.reg*W^2
122       #   for each W.  Be sure to include the 0.5 multiplying factor to
123       #   match our implementation.
124       #
125       #   And be sure to use the layers you prior implemented.
126       # ============================================================ #
127
128       loss, dLdz = softmax_loss(scores,y)
129       loss = loss+0.5*self.reg*(np.sum(W1**2)+np.sum(W2**2))
130       dh, dw2, db2 = affine_backward(dLdz,z_cache)
131       grads['W2'] = dw2 + self.reg*W2
132       grads['b2'] = db2
133
134       dx, dw1, db1 = affine_relu_backward(dh,h_cache)
135       grads['W1'] = dw1 + self.reg*W1
136       grads['b1'] = db1
137
138       # ============================================================ #
139       # END YOUR CODE HERE
140       # ============================================================ #
141
142       return loss, grads
143
144
145   class FullyConnectedNet(object):
146     """
147     A fully-connected neural network with an arbitrary number of hidden ...
          layers,
148     ReLU nonlinearities, and a softmax loss function. This will also ...
          implement
```

```
149   dropout and batch normalization as options. For a network with L layers,
150   the architecture will be
151
152   {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax
153
154   where batch normalization and dropout are optional, and the {...} ...
          block is
155   repeated L - 1 times.
156
157   Similar to the TwoLayerNet above, learnable parameters are stored in the
158   self.params dictionary and will be learned using the Solver class.
159   """
160
161   def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
162                dropout=0, use_batchnorm=False, reg=0.0,
163                weight_scale=1e-2, dtype=np.float32, seed=None):
164     """
165     Initialize a new FullyConnectedNet.
166
167     Inputs:
168     - hidden_dims: A list of integers giving the size of each hidden layer.
169     - input_dim: An integer giving the size of the input.
170     - num_classes: An integer giving the number of classes to classify.
171     - dropout: Scalar between 0 and 1 giving dropout strength. If ...
            dropout=0 then
172       the network should not use dropout at all.
173     - use_batchnorm: Whether or not the network should use batch ...
            normalization.
174     - reg: Scalar giving L2 regularization strength.
175     - weight_scale: Scalar giving the standard deviation for random
176       initialization of the weights.
177     - dtype: A numpy datatype object; all computations will be ...
            performed using
```

```
178         this datatype. float32 is faster but less accurate, so you should use
179         float64 for numeric gradient checking.
180       - seed: If not None, then pass this random seed to the dropout ...
              layers. This
181         will make the dropout layers deteriminstic so we can gradient ...
              check the
182         model.
183       """
184       self.use_batchnorm = use_batchnorm
185       self.use_dropout = dropout > 0
186       self.reg = reg
187       self.num_layers = 1 + len(hidden_dims)
188       self.dtype = dtype
189       self.params = {}
190
191       # ============================================================ #
192       # YOUR CODE HERE:
193       #   Initialize all parameters of the network in the self.params ...
              dictionary.
194       #   The weights and biases of layer 1 are W1 and b1; and in general the
195       #   weights and biases of layer i are Wi and bi. The
196       #   biases are initialized to zero and the weights are initialized
197       #   so that each parameter has mean 0 and standard deviation ...
              weight_scale.
198       #
199       #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
200       #   parameters to zero.  The gamma and beta parameters for layer 1 ...
              should
201       #   be self.params['gamma1'] and self.params['beta1'].  For layer ...
              2, they
202       #   should be gamma2 and beta2, etc. Only use batchnorm if ...
              self.use_batchnorm
203       #   is true and DO NOT do batch normalize the output scores.
```

```
204       # ============================================================ #
205
206       for i in np.arange(self.num_layers):
207         W_string = 'W' + str(i+1)
208         b_string = 'b' + str(i+1)
209         gam_str = 'gamma' + str(i+1)
210         beta_str = 'beta' + str(i+1)
211
212         #if first layer use input_dim
213         if i == 0:
214           self.params[W_string] = weight_scale * ...
                np.random.randn(input_dim, hidden_dims[i])
215           self.params[b_string] = np.zeros(hidden_dims[i])
216
217           if self.use_batchnorm:
218             self.params[gam_str] = np.ones(hidden_dims[i])
219             self.params[beta_str] = np.zeros(hidden_dims[i])
220
221         #if last layer use num_classes
222         elif i == self.num_layers - 1:
223           self.params[W_string] = weight_scale * ...
                np.random.randn(hidden_dims[i-1], num_classes)
224           self.params[b_string] = np.zeros(num_classes)
225         else:
226           self.params[W_string] = weight_scale * ...
                np.random.randn(hidden_dims[i-1], hidden_dims[i])
227           self.params[b_string] = np.zeros(hidden_dims[i])
228
229           if self.use_batchnorm:
230             self.params[gam_str] = np.ones(hidden_dims[i])
231             self.params[beta_str] = np.zeros(hidden_dims[i])
232
233
```

```
234
235
236       # ============================================================ #
237       # END YOUR CODE HERE
238       # ============================================================ #
239
240       # When using dropout we need to pass a dropout_param dictionary to each
241       # dropout layer so that the layer knows the dropout probability and ...
             the mode
242       # (train / test). You can pass the same dropout_param to each ...
             dropout layer.
243       self.dropout_param = {}
244       if self.use_dropout:
245         self.dropout_param = {'mode': 'train', 'p': dropout}
246         if seed is not None:
247           self.dropout_param['seed'] = seed
248
249       # With batch normalization we need to keep track of running means and
250       # variances, so we need to pass a special bn_param object to each batch
251       # normalization layer. You should pass self.bn_params[0] to the ...
             forward pass
252       # of the first batch normalization layer, self.bn_params[1] to the ...
             forward
253       # pass of the second batch normalization layer, etc.
254       self.bn_params = []
255       if self.use_batchnorm:
256         self.bn_params = [{'mode': 'train'} for i in ...
               np.arange(self.num_layers - 1)]
257
258       # Cast all parameters to the correct datatype
259       for k, v in self.params.items():
260         self.params[k] = v.astype(dtype)
261
```

```
262
263   def loss(self, X, y=None):
264     """
265     Compute loss and gradient for the fully-connected net.
266
267     Input / output: Same as TwoLayerNet above.
268     """
269     X = X.astype(self.dtype)
270     mode = 'test' if y is None else 'train'
271
272     # Set train/test mode for batchnorm params and dropout param since they
273     # behave differently during training and testing.
274     if self.dropout_param is not None:
275       self.dropout_param['mode'] = mode
276     if self.use_batchnorm:
277       for bn_param in self.bn_params:
278         bn_param[mode] = mode
279
280     scores = None
281
282     # ============================================================ #
283     # YOUR CODE HERE:
284     #   Implement the forward pass of the FC net and store the output
285     #   scores as the variable "scores".
286     #
287     #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
288     #   between the affine_forward and relu_forward layers.  You may
289     #   also write an affine_batchnorm_relu() function in layer_utils.py.
290     #
291     #   DROPOUT: If dropout is non-zero, insert a dropout layer after
292     #   every ReLU layer.
293     # ============================================================ #
294
```

```
295       h = []
296       h_cache = []
297       d_cache = []
298       for i in np.arange(self.num_layers):
299         W_str = 'W' + str(i+1)
300         b_str = 'b' + str(i+1)
301         gam_str = 'gamma' + str(i+1)
302         beta_str = 'beta' + str(i+1)
303
304         # if first layer
305         if i == 0:
306           if self.use_batchnorm:
307             tmp_h, tmp_h_cache = affine_batchnorm_relu_forward(X, ...
                    self.params[W_str], self.params[b_str],
308             self.params[gam_str], self.params[beta_str], self.bn_params[i])
309           else:
310             tmp_h, tmp_h_cache = ...
                    affine_relu_forward(X,self.params[W_str],self.params[b_str])
311
312           # if want to use dropout after relu layer
313           if self.use_dropout:
314             tmp_h, tmp_d_cache = dropout_forward(tmp_h, self.dropout_param)
315             d_cache.append(tmp_d_cache)
316
317           h.append(tmp_h)
318           h_cache.append(tmp_h_cache)
319
320         # last layer
321         elif i == self.num_layers - 1:
322           scores, z_cache = ...
                  affine_forward(h[i-1],self.params[W_str],self.params[b_str])
323
324         # intermediate layers
```

```
325       else:
326         if self.use_batchnorm:
327           tmp_h, tmp_h_cache = affine_batchnorm_relu_forward(h[i-1], ...
                self.params[W_str], self.params[b_str],
328           self.params[gam_str], self.params[beta_str], self.bn_params[i])
329         else:
330           tmp_h, tmp_h_cache = ...
                affine_relu_forward(h[i-1],self.params[W_str],self.params[b_str])
331
332         # if want to use dropout after relu layer
333         if self.use_dropout:
334           tmp_h, tmp_d_cache = dropout_forward(tmp_h, self.dropout_param)
335           d_cache.append(tmp_d_cache)
336
337         h.append(tmp_h)
338         h_cache.append(tmp_h_cache)
339
340     # ============================================================ #
341     # END YOUR CODE HERE
342     # ============================================================ #
343
344     # If test mode return early
345     if mode == 'test':
346       return scores
347
348     loss, grads = 0.0, {}
349     # ============================================================ #
350     # YOUR CODE HERE:
351     #   Implement the backwards pass of the FC net and store the gradients
352     #   in the grads dict, so that grads[k] is the gradient of ...
          self.params[k]
353     #   Be sure your L2 regularization includes a 0.5 factor.
354     #
```

```
355       #   BATCHNORM: Incorporate the backward pass of the batchnorm.
356       #
357       #   DROPOUT: Incorporate the backward pass of dropout.
358       # ================================================================ #
359
360       loss, dLdz = softmax_loss(scores,y)
361
362       sqr_sum = 0
363       for i in np.arange(self.num_layers,0,-1):
364         W_str = 'W' + str(i)
365         b_str = 'b' + str(i)
366         gam_str = 'gamma' + str(i)
367         beta_str = 'beta' + str(i)
368
369         sqr_sum += np.sum(self.params[W_str]**2)
370
371         # first backprop
372         if i == self.num_layers:
373           dhi , dwi, dbi = affine_backward(dLdz,z_cache)
374           grads[W_str] = dwi + self.reg*self.params[W_str]
375           grads[b_str] = dbi
376
377         # next backprop steps
378         else:
379           if self.use_dropout:
380             dhi = dropout_backward(dhi,d_cache[i-1])
381
382           if self.use_batchnorm:
383             dhi , dwi, dbi, dgami, dbetai = ...
                  affine_batchnorm_relu_backward(dhi,h_cache[i-1])
384             grads[gam_str] = dgami
385             grads[beta_str] = dbetai
386           else:
```

```
387             dhi , dwi, dbi = affine_relu_backward(dhi,h_cache[i-1])
388           grads[W_str] = dwi + self.reg*self.params[W_str]
389           grads[b_str] = dbi
390
391       loss += 0.5*self.reg*(sqr_sum)
392
393
394       # ============================================================ #
395       # END YOUR CODE HERE
396       # ============================================================ #
397
398       return loss, grads
```

A.3 layers.py

```
1  import numpy as np
2  import pdb
3
4  """
5  This code was originally written for CS 231n at Stanford University
6  (cs231n.stanford.edu).  It has been modified in various areas for use ...
       in the
7  ECE 239AS class at UCLA.  This includes the descriptions of what code to
8  implement as well as some slight potential changes in variable names to be
9  consistent with class nomenclature.  We thank Justin Johnson & Serena ...
       Yeung for
10 permission to use this code.  To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14 def affine_forward(x, w, b):
15   """
16   Computes the forward pass for an affine (fully-connected) layer.
17
18   The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
19   examples, where each example x[i] has shape (d_1, ..., d_k). We will
20   reshape each input into a vector of dimension D = d_1 * ... * d_k, and
21   then transform it to an output vector of dimension M.
22
23   Inputs:
24   - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
25   - w: A numpy array of weights, of shape (D, M)
26   - b: A numpy array of biases, of shape (M,)
27
28   Returns a tuple of:
```

```
29    - out: output, of shape (N, M)
30    - cache: (x, w, b)
31    """
32
33    # ============================================================ #
34    # YOUR CODE HERE:
35    #   Calculate the output of the forward pass.  Notice the dimensions
36    #   of w are D x M, which is the transpose of what we did in earlier
37    #   assignments.
38    # ============================================================ #
39    x_transform = x.reshape(x.shape[0],-1)
40    out = x_transform.dot(w) + b
41
42
43    # ============================================================ #
44    # END YOUR CODE HERE
45    # ============================================================ #
46
47    cache = (x, w, b)
48    return out, cache
49
50
51  def affine_backward(dout, cache):
52    """
53    Computes the backward pass for an affine layer.
54
55    Inputs:
56    - dout: Upstream derivative, of shape (N, M)
57    - cache: Tuple of:
58      - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
59      - w: A numpy array of weights, of shape (D, M)
60      - b: A numpy array of biases, of shape (M,)
61
```

```
62    Returns a tuple of:
63    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
64    - dw: Gradient with respect to w, of shape (D, M)
65    - db: Gradient with respect to b, of shape (M,)
66    """
67    x, w, b = cache
68    dx, dw, db = None, None, None
69
70    # ============================================================= #
71    # YOUR CODE HERE:
72    #   Calculate the gradients for the backward pass.
73    # Notice:
74    #   dout is N x M
75    #   dx should be N x d1 x ... x dk; it relates to dout through ...
          multiplication with w, which is D x M
76    #   dw should be D x M; it relates to dout through multiplication ...
          with x, which is N x D after reshaping
77    #   db should be M; it is just the sum over dout examples
78    # ============================================================= #
79
80    x_transform = x.reshape(x.shape[0],-1) # N x D
81    dx = dout.dot(w.T) # N x D
82    dx = dx.reshape(x.shape)
83    dw = x_transform.T.dot(dout) # D x M
84    db = np.sum(dout , axis = 0) # want to sum values in the columns to ...
          get M
85
86    # ============================================================= #
87    # END YOUR CODE HERE
88    # ============================================================= #
89
90    return dx, dw, db
91
```

```
92  def relu_forward(x):
93    """
94    Computes the forward pass for a layer of rectified linear units (ReLUs).
95
96    Input:
97    - x: Inputs, of any shape
98
99    Returns a tuple of:
100   - out: Output, of the same shape as x
101   - cache: x
102   """
103   # ============================================================ #
104   # YOUR CODE HERE:
105   #   Implement the ReLU forward pass.
106   # ============================================================ #
107
108   f = lambda x: x*(x>0)
109   out = f(x)
110
111   # ============================================================ #
112   # END YOUR CODE HERE
113   # ============================================================ #
114
115   cache = x
116   return out, cache
117
118
119 def relu_backward(dout, cache):
120   """
121   Computes the backward pass for a layer of rectified linear units (ReLUs).
122
123   Input:
124   - dout: Upstream derivatives, of any shape
```

```
125     - cache: Input x, of same shape as dout
126
127     Returns:
128     - dx: Gradient with respect to x
129     """
130     x = cache
131
132     # ============================================================ #
133     # YOUR CODE HERE:
134     #   Implement the ReLU backward pass
135     # ============================================================ #
136
137     # ReLU directs linearly to those > 0
138     dx = (x>0)*dout
139
140     # ============================================================ #
141     # END YOUR CODE HERE
142     # ============================================================ #
143
144     return dx
145
146 def batchnorm_forward(x, gamma, beta, bn_param):
147     """
148     Forward pass for batch normalization.
149
150     During training the sample mean and (uncorrected) sample variance are
151     computed from minibatch statistics and used to normalize the incoming ...
            data.
152     During training we also keep an exponentially decaying running mean ...
            of the mean
153     and variance of each feature, and these averages are used to ...
            normalize data
154     at test-time.
```

```
155
156   At each timestep we update the running averages for mean and variance ...
          using
157   an exponential decay based on the momentum parameter:
158
159   running_mean = momentum * running_mean + (1 - momentum) * sample_mean
160   running_var = momentum * running_var + (1 - momentum) * sample_var
161
162   Note that the batch normalization paper suggests a different test-time
163   behavior: they compute sample mean and variance for each feature ...
          using a
164   large number of training images rather than using a running average. For
165   this implementation we have chosen to use running averages instead since
166   they do not require an additional estimation step; the torch7 ...
          implementation
167   of batch normalization also uses running averages.
168
169   Input:
170   - x: Data of shape (N, D)
171   - gamma: Scale parameter of shape (D,)
172   - beta: Shift paremeter of shape (D,)
173   - bn_param: Dictionary with the following keys:
174     - mode: 'train' or 'test'; required
175     - eps: Constant for numeric stability
176     - momentum: Constant for running mean / variance.
177     - running_mean: Array of shape (D,) giving running mean of features
178     - running_var Array of shape (D,) giving running variance of features
179
180   Returns a tuple of:
181   - out: of shape (N, D)
182   - cache: A tuple of values needed in the backward pass
183   """
184   mode = bn_param['mode']
```

```
185   eps = bn_param.get('eps', 1e-5)
186   momentum = bn_param.get('momentum', 0.9)
187
188   N, D = x.shape
189   running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
190   running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
191
192   out, cache = None, None
193   if mode == 'train':
194
195     # ============================================================ #
196     # YOUR CODE HERE:
197     #   A few steps here:
198     #     (1) Calculate the running mean and variance of the minibatch.
199     #     (2) Normalize the activations with the running mean and variance.
200     #     (3) Scale and shift the normalized activations.  Store this
201     #         as the variable 'out'
202     #     (4) Store any variables you may need for the backward pass in
203     #         the 'cache' variable.
204     # ============================================================ #
205     running_mean = 1/N*np.sum(x, axis = 0) #(D,)
206     running_var = 1/N*np.sum((x-running_mean)**2, axis = 0) #(D,)
207
208     xhat = (x - running_mean)/np.sqrt(running_var + eps) #(N,D)
209     out = gamma*xhat + beta
210
211     cache = (x, xhat, gamma, running_mean, running_var, eps)
212
213     # ============================================================ #
214     # END YOUR CODE HERE
215     # ============================================================ #
216   elif mode == 'test':
217
```

```
218       # ============================================================ #
219       # YOUR CODE HERE:
220       #   Calculate the testing time normalized activation.  Normalize using
221       #   the running mean and variance, and then scale and shift ...
              appropriately.
222       #   Store the output as 'out'.
223       # ============================================================ #
224       xhat = (x - running_mean)/np.sqrt(running_var + eps)
225       out = gamma*xhat + beta
226
227       # ============================================================ #
228       # END YOUR CODE HERE
229       # ============================================================ #
230
231     else:
232       raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
233
234     # Store the updated running means back into bn_param
235     bn_param['running_mean'] = running_mean
236     bn_param['running_var'] = running_var
237
238     return out, cache
239
240   def batchnorm_backward(dout, cache):
241     """
242     Backward pass for batch normalization.
243
244     For this implementation, you should write out a computation graph for
245     batch normalization on paper and propagate gradients backward through
246     intermediate nodes.
247
248     Inputs:
249     - dout: Upstream derivatives, of shape (N, D)
```

```
250     - cache: Variable of intermediates from batchnorm_forward.
251
252     Returns a tuple of:
253     - dx: Gradient with respect to inputs x, of shape (N, D)
254     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
255     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
256     """
257     dx, dgamma, dbeta = None, None, None
258
259     # ============================================================== #
260     # YOUR CODE HERE:
261     #   Implement the batchnorm backward pass, calculating dx, dgamma, ...
            and dbeta.
262     # ============================================================== #
263     x, xhat, gamma, mu, var, eps = cache
264     # sizes
265     # x (N,D); xhat (N,D); gamma (D,); var (D,): mu (D,); eps (1,)
266     N, D = dout.shape
267
268     # summing along columns
269     dbeta = np.sum(dout,axis = 0) #(D,)
270     dgamma = np.sum(dout*xhat,axis = 0) #(D,)
271
272     dxhat = dout*gamma
273
274     dvar = np.sum(-1/(2*(var + eps)**(3/2))*(x-mu)*dxhat, axis = 0) #(D,)
275     # a = x - mu
276     da = 1/np.sqrt(var+eps)*dxhat #(N,D)
277
278     # law of total derivatives for mean
279     dvardu = -2*(x-mu)/N
280     dmu = np.sum(-da + dvardu*dvar, axis = 0) #(D,)
281
```

```
282   # law of total derivatives for x
283   dx = da + 2*(x-mu)/N*dvar + 1/N*dmu
284
285   # ============================================================== #
286   # END YOUR CODE HERE
287   # ============================================================== #
288
289   return dx, dgamma, dbeta
290
291 def dropout_forward(x, dropout_param):
292   """
293   Performs the forward pass for (inverted) dropout.
294
295   Inputs:
296   - x: Input data, of any shape
297   - dropout_param: A dictionary with the following keys:
298     - p: Dropout parameter. We keep each neuron output with probability p.
299     - mode: 'test' or 'train'. If the mode is train, then perform dropout;
300       if the mode is test, then just return the input.
301     - seed: Seed for the random number generator. Passing seed makes this
302       function deterministic, which is needed for gradient checking but ...
              not in
303       real networks.
304
305   Outputs:
306   - out: Array of the same shape as x.
307   - cache: A tuple (dropout_param, mask). In training mode, mask is the ...
        dropout
308     mask that was used to multiply the input; in test mode, mask is None.
309   """
310   p, mode = dropout_param['p'], dropout_param['mode']
311   if 'seed' in dropout_param:
312     np.random.seed(dropout_param['seed'])
```

```
313
314     mask = None
315     out = None
316
317     if mode == 'train':
318       # ============================================================ #
319       # YOUR CODE HERE:
320       #   Implement the inverted dropout forward pass during training time.
321       #   Store the masked and scaled activations in out, and store the
322       #   dropout mask as the variable mask.
323       # ============================================================ #
324       mask = (np.random.rand(*x.shape) < p) /p
325       out = x*mask
326
327       # ============================================================ #
328       # END YOUR CODE HERE
329       # ============================================================ #
330
331     elif mode == 'test':
332
333       # ============================================================ #
334       # YOUR CODE HERE:
335       #   Implement the inverted dropout forward pass during test time.
336       # ============================================================ #
337       out = x
338       # ============================================================ #
339       # END YOUR CODE HERE
340       # ============================================================ #
341
342     cache = (dropout_param, mask)
343     out = out.astype(x.dtype, copy=False)
344
345     return out, cache
```

```
346
347 def dropout_backward(dout, cache):
348   """
349   Perform the backward pass for (inverted) dropout.
350
351   Inputs:
352   - dout: Upstream derivatives, of any shape
353   - cache: (dropout_param, mask) from dropout_forward.
354   """
355   dropout_param, mask = cache
356   mode = dropout_param['mode']
357
358   dx = None
359   if mode == 'train':
360     # ============================================================ #
361     # YOUR CODE HERE:
362     #   Implement the inverted dropout backward pass during training time.
363     # ============================================================ #
364     dx = dout*mask
365     # ============================================================ #
366     # END YOUR CODE HERE
367     # ============================================================ #
368
369   elif mode == 'test':
370     # ============================================================ #
371     # YOUR CODE HERE:
372     #   Implement the inverted dropout backward pass during test time.
373     # ============================================================ #
374     dx = dout
375     # ============================================================ #
376     # END YOUR CODE HERE
377     # ============================================================ #
378   return dx
```

```
379
380 def svm_loss(x, y):
381   """
382   Computes the loss and gradient using for multiclass SVM classification.
383
384   Inputs:
385   - x: Input data, of shape (N, C) where x[i, j] is the score for the ...
          jth class
386     for the ith input.
387   - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
388     0 ≤ y[i] < C
389
390   Returns a tuple of:
391   - loss: Scalar giving the loss
392   - dx: Gradient of the loss with respect to x
393   """
394   N = x.shape[0]
395   correct_class_scores = x[np.arange(N), y]
396   margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
397   margins[np.arange(N), y] = 0
398   loss = np.sum(margins) / N
399   num_pos = np.sum(margins > 0, axis=1)
400   dx = np.zeros_like(x)
401   dx[margins > 0] = 1
402   dx[np.arange(N), y] -= num_pos
403   dx /= N
404   return loss, dx
405
406
407 def softmax_loss(x, y):
408   """
409   Computes the loss and gradient for softmax classification.
410
```

```
411   Inputs:
412   - x: Input data, of shape (N, C) where x[i, j] is the score for the ...
          jth class
413     for the ith input.
414   - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
415     0 ≤ y[i] < C
416
417   Returns a tuple of:
418   - loss: Scalar giving the loss
419   - dx: Gradient of the loss with respect to x
420   """
421
422   probs = np.exp(x - np.max(x, axis=1, keepdims=True))
423   probs /= np.sum(probs, axis=1, keepdims=True)
424   N = x.shape[0]
425   loss = -np.sum(np.log(probs[np.arange(N), y])) / N
426   dx = probs.copy()
427   dx[np.arange(N), y] -= 1
428   dx /= N
429   return loss, dx
```

A.4 layer_utils.py

```
from .layers 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.
"""

def affine_relu_forward(x, w, b):
  """
  Convenience layer that performs an affine transform followed by a ReLU

  Inputs:
  - x: Input to the affine layer
  - w, b: Weights for the affine layer

  Returns a tuple of:
  - out: Output from the ReLU
  - cache: Object to give to the backward pass
  """
  a, fc_cache = affine_forward(x, w, b)
  out, relu_cache = relu_forward(a)
  cache = (fc_cache, relu_cache)
  return out, cache
```

```
29
30
31 def affine_relu_backward(dout, cache):
32   """
33   Backward pass for the affine-relu convenience layer
34   """
35   fc_cache, relu_cache = cache
36   da = relu_backward(dout, relu_cache)
37   dx, dw, db = affine_backward(da, fc_cache)
38   return dx, dw, db
39
40 def affine_batchnorm_relu_forward(x, w, b, gamma, beta, bn_param):
41   """
42   Convenience layer that performs an affine transform followed by ...
       batchnorm followed by a ReLU
43
44   Inputs:
45   - x: Input to the affine layer
46   - w, b: Weights for the affine layer
47
48   Returns a tuple of:
49   - out: Output from the ReLU
50   - cache: Object to give to the backward pass
51   """
52   a, fc_cache = affine_forward(x, w, b)
53   c, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
54   out, relu_cache = relu_forward(c)
55   cache = (fc_cache, bn_cache, relu_cache)
56   return out, cache
57
58 def affine_batchnorm_relu_backward(dout, cache):
59   """
60   Backward pass for the affine-batchnorm-relu convenience layer
```

```
    """
    fc_cache, bn_cache, relu_cache = cache
    dc = relu_backward(dout, relu_cache)
    da, dgamma, dbeta = batchnorm_backward(dc, bn_cache)
    dx, dw, db = affine_backward(da, fc_cache)

    return dx, dw, db, dgamma, dbeta
```