Optimization

February 9, 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).

[9]:
```
## 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))))
```

[11]:
```
# 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

[12]:
```
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: 9.769847728806635e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 2.976840298156737e-09
dw error: 2.393432626289272e-10
db error: 6.752845919577194e-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: 3.2756066015352925e-12


If affine_relu_forward and affine_relu_backward are working, error should be
less than 1e-9::
dx error: 7.387396782814579e-11
dw error: 1.8158063144465535e-09
db error: 1.4870265928801125e-11


Running check with reg = 0
Initial loss: 2.3047954164728917
W1 relative error: 3.2772426764578215e-07
W2 relative error: 7.968430874024777e-06
W3 relative error: 5.929495818249865e-08
b1 relative error: 1.3499731214174142e-08
b2 relative error: 4.7763758632431615e-09
b3 relative error: 1.866906974850144e-10
Running check with reg = 3.14
Initial loss: 7.004724143938643
W1 relative error: 7.862362479357505e-09
W2 relative error: 1.2872025349112612e-08
W3 relative error: 1.6852169609719076e-08
b1 relative error: 1.2970303036711917e-08
b2 relative error: 4.315547899837167e-09
b3 relative error: 7.824371579799215e-11
```

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.

[14]:
```
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`.

[15]:
```
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: 1.0875187099974104e-08
velocity error: 4.269287743278663e-09
```

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

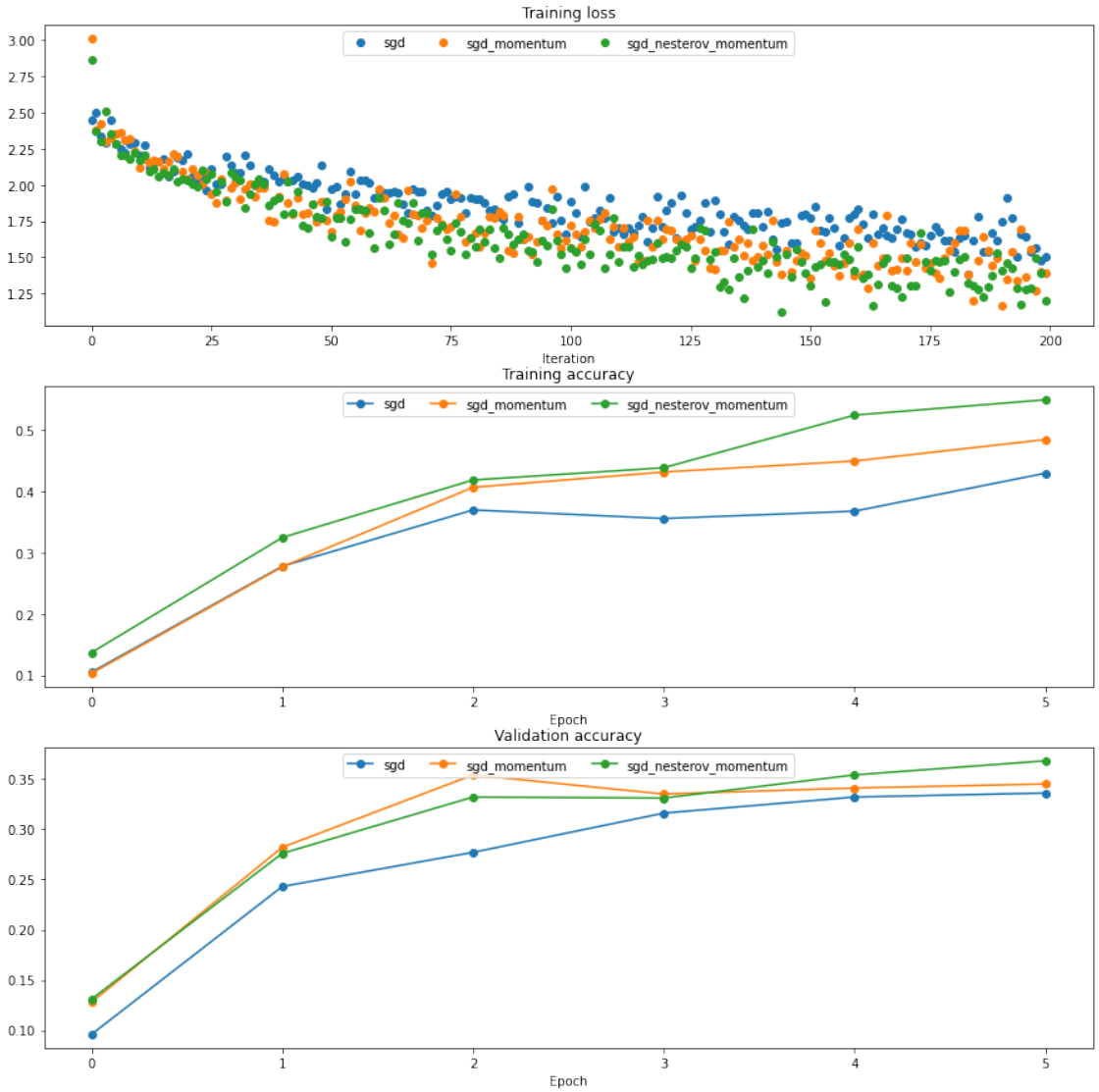
Run the following cell to train a 6 layer FC net with SGD, SGD+momentum, and SGD+Nesterov momentum. You should see that SGD+momentum achieves a better loss than SGD, and that SGD+Nesterov momentum achieves a slightly better loss (and training accuracy) than SGD+momentum.

[16]:
```
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
```

```
<ipython-input-16-52ffd8523dca>:39: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 1)
<ipython-input-16-52ffd8523dca>:42: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-16-52ffd8523dca>:45: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-16-52ffd8523dca>:49: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, i)
```



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.

[22]:
```
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.

[29]:
```
# 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

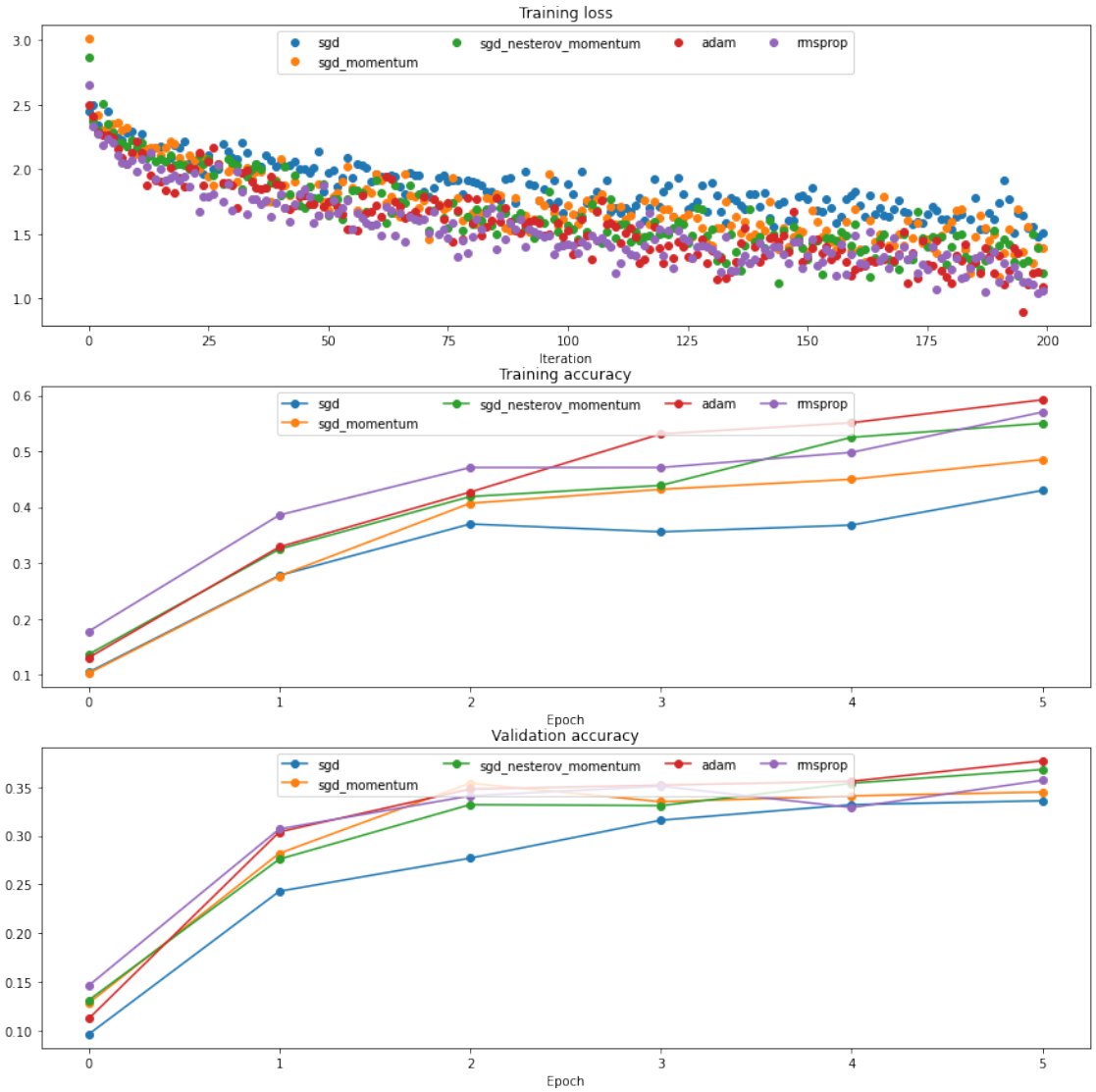
The following code will compare optimization with SGD, Momentum, Nesterov Momentum, RMSProp and Adam. In our code, we find that RMSProp, Adam, and SGD + Nesterov Momentum achieve approximately the same training error after a few training epochs.

[30]:
```
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
```

```
<ipython-input-30-27795ef4623c>:31: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 1)
<ipython-input-30-27795ef4623c>:34: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-30-27795ef4623c>:37: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-30-27795ef4623c>:41: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, i)
```



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.

[31]:
```
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: 2.299246
(Epoch 0 / 10) train acc: 0.172000; val_acc: 0.159000
(Iteration 51 / 4900) loss: 1.997704
(Iteration 101 / 4900) loss: 1.861022
(Iteration 151 / 4900) loss: 1.740768
(Iteration 201 / 4900) loss: 1.592706
(Iteration 251 / 4900) loss: 1.741350
(Iteration 301 / 4900) loss: 1.632480
(Iteration 351 / 4900) loss: 1.671803
(Iteration 401 / 4900) loss: 1.757504
(Iteration 451 / 4900) loss: 1.825510
(Epoch 1 / 10) train acc: 0.463000; val_acc: 0.447000
(Iteration 501 / 4900) loss: 1.615169
(Iteration 551 / 4900) loss: 1.404278
(Iteration 601 / 4900) loss: 1.475674
(Iteration 651 / 4900) loss: 1.534499
(Iteration 701 / 4900) loss: 1.610334
(Iteration 751 / 4900) loss: 1.486369
(Iteration 801 / 4900) loss: 1.606933
(Iteration 851 / 4900) loss: 1.654892
(Iteration 901 / 4900) loss: 1.273844
(Iteration 951 / 4900) loss: 1.249665
(Epoch 2 / 10) train acc: 0.488000; val_acc: 0.446000
(Iteration 1001 / 4900) loss: 1.491296
(Iteration 1051 / 4900) loss: 1.337868
(Iteration 1101 / 4900) loss: 1.282175
(Iteration 1151 / 4900) loss: 1.462374
(Iteration 1201 / 4900) loss: 1.542706
(Iteration 1251 / 4900) loss: 1.447951
(Iteration 1301 / 4900) loss: 1.469194
(Iteration 1351 / 4900) loss: 1.408071
(Iteration 1401 / 4900) loss: 1.339516
(Iteration 1451 / 4900) loss: 1.439425
```

```
(Epoch 3 / 10) train acc: 0.509000; val_acc: 0.496000
(Iteration 1501 / 4900) loss: 1.285854
(Iteration 1551 / 4900) loss: 1.364739
(Iteration 1601 / 4900) loss: 1.500294
(Iteration 1651 / 4900) loss: 1.458643
(Iteration 1701 / 4900) loss: 1.210685
(Iteration 1751 / 4900) loss: 1.270577
(Iteration 1801 / 4900) loss: 1.250438
(Iteration 1851 / 4900) loss: 1.305539
(Iteration 1901 / 4900) loss: 1.480247
(Iteration 1951 / 4900) loss: 1.399973
(Epoch 4 / 10) train acc: 0.508000; val_acc: 0.505000
(Iteration 2001 / 4900) loss: 1.303328
(Iteration 2051 / 4900) loss: 1.502038
(Iteration 2101 / 4900) loss: 1.508286
(Iteration 2151 / 4900) loss: 1.114392
(Iteration 2201 / 4900) loss: 1.197692
(Iteration 2251 / 4900) loss: 1.266051
(Iteration 2301 / 4900) loss: 1.054477
(Iteration 2351 / 4900) loss: 1.307056
(Iteration 2401 / 4900) loss: 1.342271
(Epoch 5 / 10) train acc: 0.574000; val_acc: 0.507000
(Iteration 2451 / 4900) loss: 1.281585
(Iteration 2501 / 4900) loss: 1.219263
(Iteration 2551 / 4900) loss: 1.138957
(Iteration 2601 / 4900) loss: 1.191536
(Iteration 2651 / 4900) loss: 1.228345
(Iteration 2701 / 4900) loss: 1.279234
(Iteration 2751 / 4900) loss: 1.260086
(Iteration 2801 / 4900) loss: 0.907153
(Iteration 2851 / 4900) loss: 1.210876
(Iteration 2901 / 4900) loss: 1.164848
(Epoch 6 / 10) train acc: 0.558000; val_acc: 0.519000
(Iteration 2951 / 4900) loss: 1.185043
(Iteration 3001 / 4900) loss: 0.969132
(Iteration 3051 / 4900) loss: 1.241063
(Iteration 3101 / 4900) loss: 1.301104
(Iteration 3151 / 4900) loss: 1.310557
(Iteration 3201 / 4900) loss: 1.256410
(Iteration 3251 / 4900) loss: 1.122481
(Iteration 3301 / 4900) loss: 1.079146
(Iteration 3351 / 4900) loss: 1.239904
(Iteration 3401 / 4900) loss: 0.942272
(Epoch 7 / 10) train acc: 0.619000; val_acc: 0.512000
(Iteration 3451 / 4900) loss: 0.881264
(Iteration 3501 / 4900) loss: 0.906770
(Iteration 3551 / 4900) loss: 1.342381
(Iteration 3601 / 4900) loss: 1.096470
```

```
(Iteration 3651 / 4900) loss: 0.874140
(Iteration 3701 / 4900) loss: 0.899245
(Iteration 3751 / 4900) loss: 1.188415
(Iteration 3801 / 4900) loss: 1.103673
(Iteration 3851 / 4900) loss: 0.967107
(Iteration 3901 / 4900) loss: 0.905143
(Epoch 8 / 10) train acc: 0.626000; val_acc: 0.526000
(Iteration 3951 / 4900) loss: 1.121037
(Iteration 4001 / 4900) loss: 1.101415
(Iteration 4051 / 4900) loss: 1.282406
(Iteration 4101 / 4900) loss: 1.044541
(Iteration 4151 / 4900) loss: 0.977709
(Iteration 4201 / 4900) loss: 1.107077
(Iteration 4251 / 4900) loss: 0.965687
(Iteration 4301 / 4900) loss: 0.885222
(Iteration 4351 / 4900) loss: 1.165078
(Iteration 4401 / 4900) loss: 1.060734
(Epoch 9 / 10) train acc: 0.645000; val_acc: 0.542000
(Iteration 4451 / 4900) loss: 1.014575
(Iteration 4501 / 4900) loss: 0.769887
(Iteration 4551 / 4900) loss: 0.880520
(Iteration 4601 / 4900) loss: 1.005056
(Iteration 4651 / 4900) loss: 0.918810
(Iteration 4701 / 4900) loss: 0.853304
(Iteration 4751 / 4900) loss: 1.022516
(Iteration 4801 / 4900) loss: 0.874242
(Iteration 4851 / 4900) loss: 1.126077
(Epoch 10 / 10) train acc: 0.694000; val_acc: 0.555000
```

[32]:
```
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: 0.555
Test set accuracy: 0.563
```

Just want to point out that running all 10 epochs took me close to 40 minutes .. really glad that I got it the first time! yikes

Now, here's my optim.py in the cell below.

[ ]:
```
import numpy as np

"""
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.
"""

"""
This file implements various first-order update rules that are commonly used for
training neural networks. Each update rule accepts current weights and the
gradient of the loss with respect to those weights and produces the next set of
weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:
  - w: A numpy array giving the current weights.
  - dw: A numpy array of the same shape as w giving the gradient of the
    loss with respect to w.
  - config: A dictionary containing hyperparameter values such as learning rate,
    momentum, etc. If the update rule requires caching values over many
    iterations, then config will also hold these cached values.

Returns:
  - next_w: The next point after the update.
  - config: The config dictionary to be passed to the next iteration of the
    update rule.

NOTE: For most update rules, the default learning rate will probably not perform
well; however the default values of the other hyperparameters should work well
for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w and
setting next_w equal to w.
"""


def sgd(w, dw, config=None):
  """
  Performs vanilla stochastic gradient descent.

  config format:
  - learning_rate: Scalar learning rate.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)

  w -= config['learning_rate'] * dw
```

```
  return w, config


def sgd_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))         # gets velocity, else␣
↪sets it to zero.

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and the updated velocity as v.
  # ================================================================ #

  #adapted from lecture 10 slides, the slide titled "Momentum", also page 296␣
↪of the deep learning textbook
  v = config['momentum']*v - config['learning_rate']*dw
  next_w = w + v

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

  config['velocity'] = v

  return next_w, config

def sgd_nesterov_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with Nesterov momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
```

```
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))         # gets velocity, else␣
↪sets it to zero.

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and the updated velocity as v.
  # ================================================================ #

  #using https://cs231n.github.io/neural-networks-3/#sgd
  v_prev = v
  v = config['momentum'] * v - config['learning_rate']*dw
  next_w = w - config['momentum'] * v_prev + (1 + config['momentum']) * v

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

  config['velocity'] = v

  return next_w, config

def rmsprop(w, dw, config=None):
  """
  Uses the RMSProp update rule, which uses a moving average of squared gradient
  values to set adaptive per-parameter learning rates.

  config format:
  - learning_rate: Scalar learning rate.
  - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
    gradient cache.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - beta: Moving average of second moments of gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('decay_rate', 0.99)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('a', np.zeros_like(w))

  next_w = None
```

```
  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement RMSProp.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, so they can be used for future gradients. Concretely,
  #   config['a'] corresponds to "a" in the lecture notes.
  # ================================================================ #

  #using https://cs231n.github.io/neural-networks-3/#sgd as a guide, along w␣
 ↪lecture slides
  #a = cache
  config['a'] = config['decay_rate'] * config['a'] + (1 - config['decay_rate'])␣
 ↪* dw**2

  #in this case, epsilon represents epsilon, not learning rate
  next_w = w - (config['learning_rate'] * dw / (np.sqrt(config['a']) +␣
 ↪config['epsilon']))

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

  return next_w, config


def adam(w, dw, config=None):
  """
  Uses the Adam update rule, which incorporates moving averages of both the
  gradient and its square and a bias correction term.

  config format:
  - learning_rate: Scalar learning rate.
  - beta1: Decay rate for moving average of first moment of gradient.
  - beta2: Decay rate for moving average of second moment of gradient.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - m: Moving average of gradient.
  - v: Moving average of squared gradient.
  - t: Iteration number.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-3)
  config.setdefault('beta1', 0.9)
  config.setdefault('beta2', 0.999)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('v', np.zeros_like(w))
  config.setdefault('a', np.zeros_like(w))
```

```
config.setdefault('t', 0)

next_w = None

# ============================================================ #
# YOUR CODE HERE:
#   Implement Adam.  Store the next value of w as next_w.  You need
#   to also store in config['a'] the moving average of the second
#   moment gradients, and in config['v'] the moving average of the
#   first moments.  Finally, store in config['t'] the increasing time.
# ============================================================ #

config['t'] += 1

#m -> config['v'], v -> config['a']
config['v'] = config['beta1']*config['v'] + (1-config['beta1']) * dw
config['a'] = config['beta2']*config['a'] + (1-config['beta2'])* dw * dw

#bias correction (I tried w/o bias correction and I had a lot of error)

vt = config['v'] / (1 - config['beta1']**config['t'])
at = config['a'] / (1 - config['beta2']**config['t'])

next_w = w - config['learning_rate'] * vt / (np.sqrt(at) + config['epsilon'])

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

return next_w, config
```

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).

[1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from nndl.layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

[2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

1.1 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.

[12]:
```
# 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:  [ 8.59269312 -5.23135849 26.86727096]
```

```
  stds:  [32.94804157 30.64593955 30.65345025]
After batch normalization (gamma=1, beta=0)
  mean:  [-7.10542736e-17  4.30211422e-17  1.60982339e-16]
  std:  [1.         0.99999999 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.

[13]:
```
# 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:  [-0.04389632  0.02964754  0.0984425 ]
  stds:  [0.99818918 0.94906423 1.1064687 ]
```

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.

[21]:
```
# 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:  1.1248049066356201e-09
dgamma error:  1.3401129392003245e-11
dbeta error:  3.275629256263503e-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.

[30]:
```
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:  2.305933157828989
W1 relative error: 7.7477420586017e-05
W2 relative error: 4.196165108723515e-06
W3 relative error: 4.560476976762307e-10
b1 relative error: 0.002221689499037893
b2 relative error: 1.2434497875801753e-06
b3 relative error: 1.20353142917276e-10
beta1 relative error: 9.529613661272138e-09
beta2 relative error: 9.025460213519346e-09
gamma1 relative error: 7.124807552229981e-09
gamma2 relative error: 4.016190136037638e-09


Running check with reg =  3.14
Initial loss:  7.215262772233972
W1 relative error: 1.5879900801850465e-07
W2 relative error: 1.2270139496151539e-05
W3 relative error: 4.642431841796509e-09
b1 relative error: 4.163336342344337e-09
b2 relative error: 8.881784197001252e-08
b3 relative error: 3.8085525338151507e-10
beta1 relative error: 3.214152745609324e-08
beta2 relative error: 1.2542086077580079e-08
gamma1 relative error: 3.715365067200385e-08
gamma2 relative error: 2.2708724620180818e-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.

[31]:
```
# 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()
```

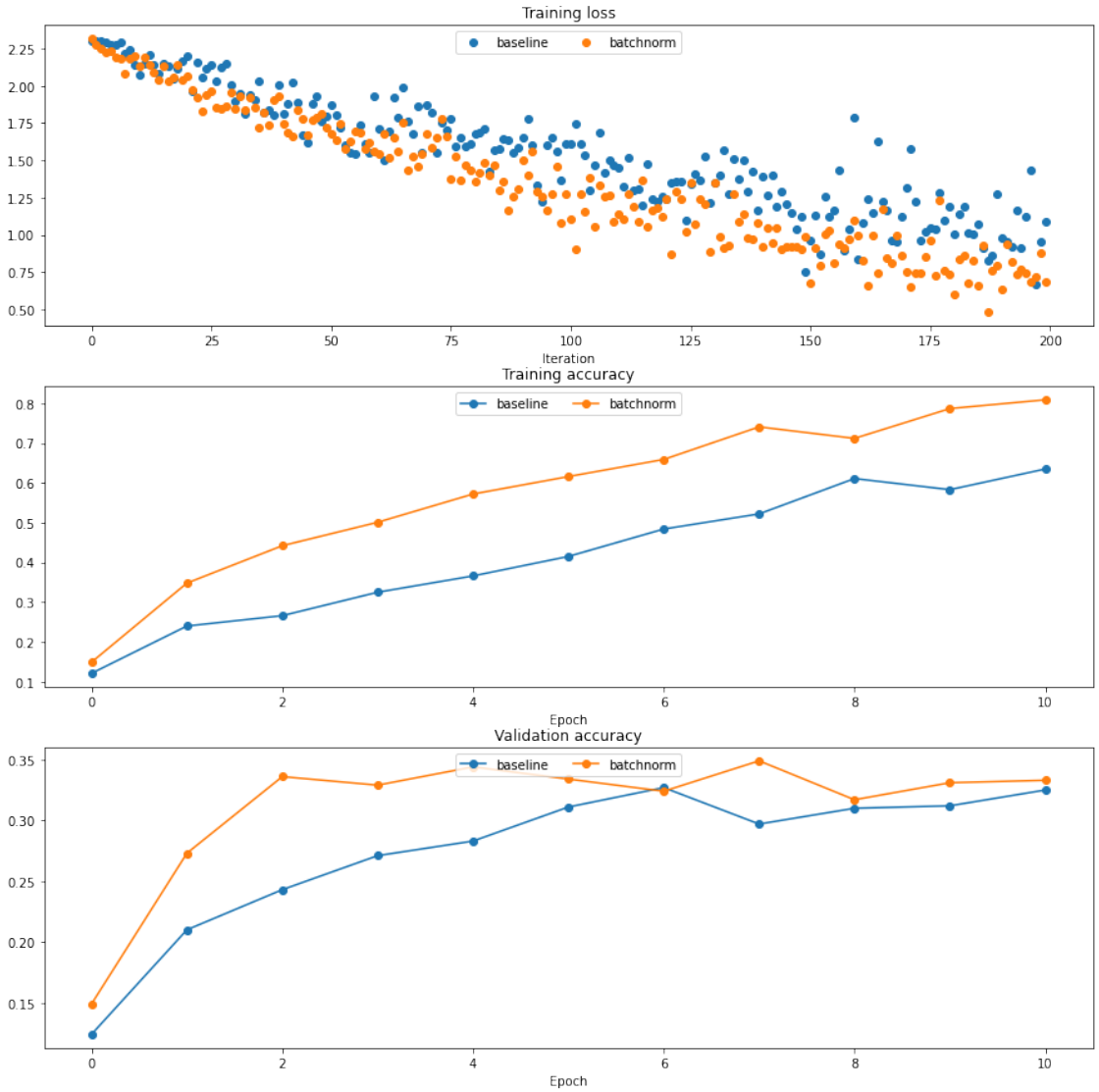
```
(Iteration 1 / 200) loss: 2.315647
(Epoch 0 / 10) train acc: 0.149000; val_acc: 0.149000
(Epoch 1 / 10) train acc: 0.348000; val_acc: 0.273000
(Epoch 2 / 10) train acc: 0.442000; val_acc: 0.336000
(Epoch 3 / 10) train acc: 0.501000; val_acc: 0.329000
(Epoch 4 / 10) train acc: 0.572000; val_acc: 0.344000
(Epoch 5 / 10) train acc: 0.616000; val_acc: 0.334000
(Epoch 6 / 10) train acc: 0.659000; val_acc: 0.324000
(Epoch 7 / 10) train acc: 0.741000; val_acc: 0.349000
(Epoch 8 / 10) train acc: 0.712000; val_acc: 0.317000
(Epoch 9 / 10) train acc: 0.787000; val_acc: 0.331000
(Epoch 10 / 10) train acc: 0.809000; val_acc: 0.333000
```

```
(Iteration 1 / 200) loss: 2.302030
(Epoch 0 / 10) train acc: 0.121000; val_acc: 0.124000
(Epoch 1 / 10) train acc: 0.240000; val_acc: 0.210000
(Epoch 2 / 10) train acc: 0.266000; val_acc: 0.243000
(Epoch 3 / 10) train acc: 0.325000; val_acc: 0.271000
(Epoch 4 / 10) train acc: 0.366000; val_acc: 0.283000
(Epoch 5 / 10) train acc: 0.415000; val_acc: 0.311000
(Epoch 6 / 10) train acc: 0.484000; val_acc: 0.327000
(Epoch 7 / 10) train acc: 0.522000; val_acc: 0.297000
(Epoch 8 / 10) train acc: 0.611000; val_acc: 0.310000
(Epoch 9 / 10) train acc: 0.583000; val_acc: 0.312000
(Epoch 10 / 10) train acc: 0.635000; val_acc: 0.325000
```

[32]:
```
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()
```

```
<ipython-input-32-8e49aa315b6d>:13: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
```

```
  plt.subplot(3, 1, 1)
<ipython-input-32-8e49aa315b6d>:17: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-32-8e49aa315b6d>:21: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-32-8e49aa315b6d>:26: MatplotlibDeprecationWarning: Adding an axes
using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  plt.subplot(3, 1, i)
```



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.

[33]:
```
# 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'],
```

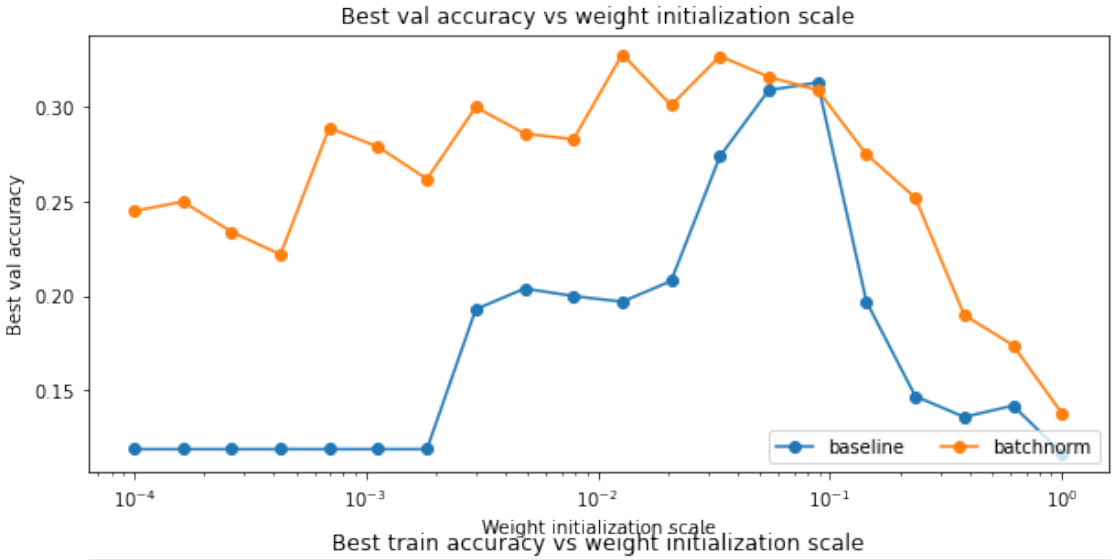
```
  'y_val': data['y_val'],
}



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
```

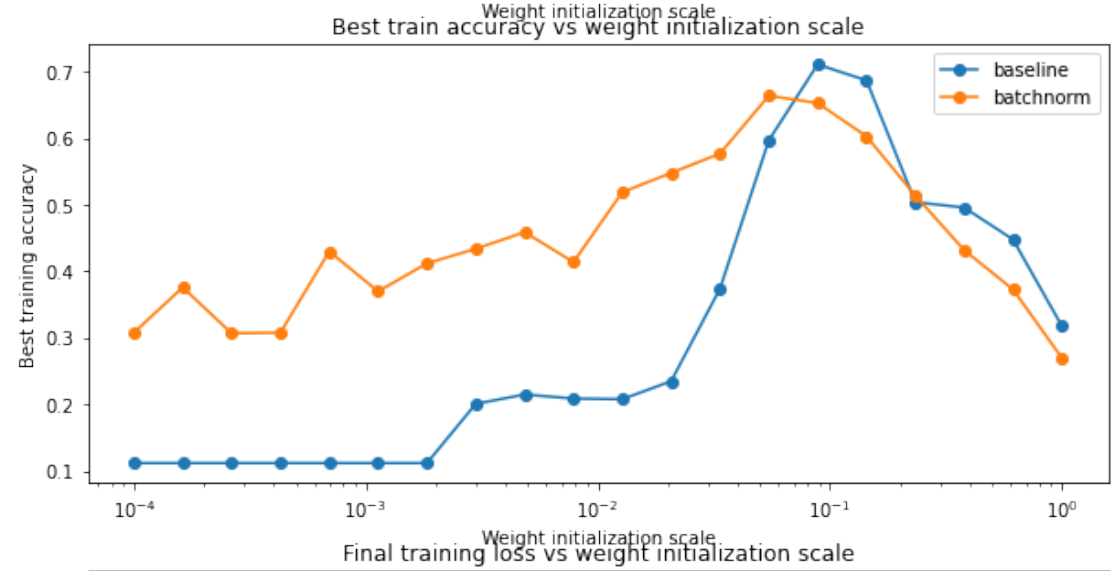
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale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

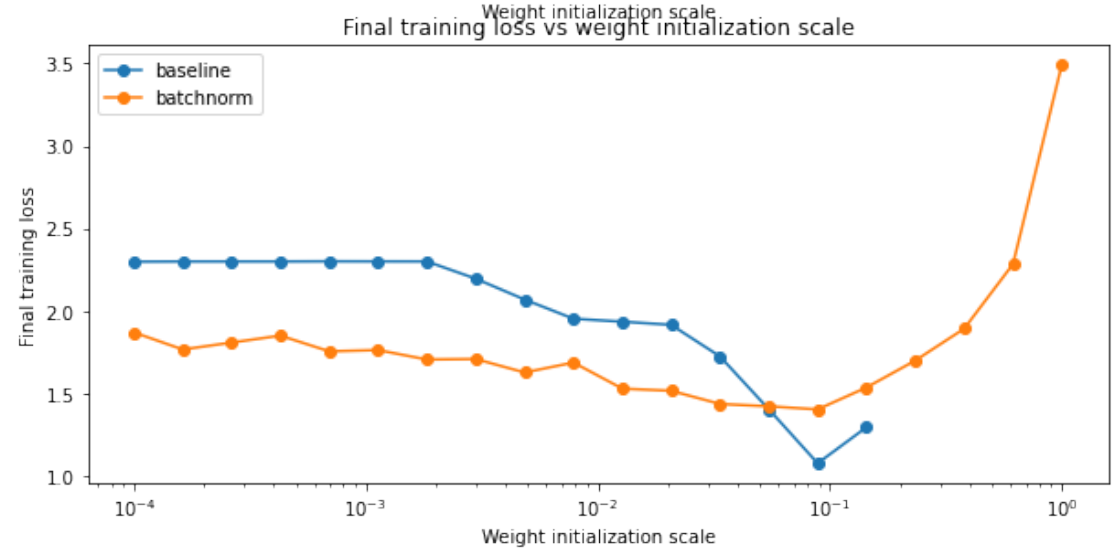
```
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20

C:\Users\Ashwin\Desktop\UCLA\current classes\ece
247\hw4\HW4-code\nndl\layers.py:444: RuntimeWarning: divide by zero encountered
in log
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N

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

[34]:
```
# 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
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
3.5
3.0
2.5
2.0
1.5
1.0
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:

(note there's a divide by 0 error, and adding a small epsilon value to the softmax function should solve the problem. However my graphs looked correct regardless, so I figured this wasn't necessary in my case)

The whole point of batchnorm is to reduce extreme changes based on initial weights propogating through the hidden layers. Hence, we expect the batchnorm neural network to be more resistant to change than the baseline neural network due to different initial weights.

Indeed, this is what we see. In particular, batchnorm prevents explosion (weights going to infinity), and vanishing (weights going to 0); however, it looks like ~10^-3, the baseline model's weights started to vanish, leading to low training accuracy, and similarly around 10^0 = 1 for exploding weights. Note the batchnorm neural network is resistant to these low/high weights due to its enforcement of unit variance + zero mean.

Additionally, gradient descent converges faster (less zigzagging) when batchnorm is employed, hence leading to higher performance; we can see that batchnorm mostly outperforms the baseline model over all weights.

1.7.1 fc_net.py:

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

from .layers import *
from .layer_utils import *

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

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

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

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

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

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

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

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ============================================================ #
    self.params['W1'] = np.random.normal(scale=weight_scale, size = (input_dim,␣
↪hidden_dims), loc=0.0)
    self.params['W2'] = np.random.normal(scale=weight_scale, size =␣
↪(hidden_dims, num_classes), loc=0.0)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['b2'] = np.zeros(num_classes)

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

  def loss(self, X, y=None):
    """
```

```
    Compute loss and gradient for a minibatch of data.

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

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

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

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ============================================================ #
    W1, W2, b1, b2 = self.params['W1'], self.params['W2'], self.params['b1'],␣
↪self.params['b2']

    hidden, hidden_cache = affine_relu_forward(X, W1, b1)
    scores, scores_cache = affine_forward(hidden, W2, b2) #forward prop
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

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

    loss, grads = 0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Store
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
```

```
    #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ============================================================ #
    loss, dscores = softmax_loss(scores, y)

    sum_term = np.sum(W1*W1) + np.sum(W2*W2)
    reg_term = 0.5 * self.reg * sum_term
    loss += reg_term

    dhidden, dW2, db2 = affine_backward(dscores, scores_cache)
    dX, dW1, db1 = affine_relu_backward(dhidden, hidden_cache)

    #update gradient

    grads['W1'] = self.reg * W1 + dW1
    grads['W2'] = self.reg * W2 + dW2
    grads['b1'] = db1
    grads['b2'] = db2
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

    return loss, grads


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

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

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

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

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

```
    """
    Initialize a new FullyConnectedNet.

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

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params dictionary.
    #   The weights and biases of layer 1 are W1 and b1; and in general the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #
    #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
    #   parameters to zero.  The gamma and beta parameters for layer 1 should
    #   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
    #   should be gamma2 and beta2, etc. Only use batchnorm if self.
↪use_batchnorm
    #   is true and DO NOT do batch normalize the output scores.
    # ============================================================ #
    #start our loop here, going through all the layers
    for i in range(1, self.num_layers + 1):

        #add batchnorm
```

```
        if self.use_batchnorm and i != self.num_layers: #if i = self.
↪num_layers, index is out of range?
            gam = "gamma" + str(i)
            beta = "beta" + str(i)
            self.params[gam] = np.ones(hidden_dims[i-1])
            self.params[beta] = np.zeros(hidden_dims[i-1]) #0, 1 wouldn't work␣
↪lol

        #make our layers' names
        W = 'W' + str(i) #'W' + i wouldn't work
        b = 'b' + str(i)

        size, zeros_size = 0, 0

        #last layer
        if i == self.num_layers:
            size = (hidden_dims[i-2], num_classes)
            zeros_size = num_classes

        #first layer
        elif i == 1:
            size = (input_dim, hidden_dims[i-1])
            zeros_size = hidden_dims[i-1]

        #hidden layers
        else:
            size = (hidden_dims[i-2], hidden_dims[i-1])
            zeros_size = hidden_dims[i-1]

        #update params
        self.params[W] = np.random.normal(loc=0.0, scale=weight_scale,␣
↪size=size)
        self.params[b] = np.zeros(zeros_size)

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

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

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

    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


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

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

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

    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    #
    #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
    #   between the affine_forward and relu_forward layers.  You may
    #   also write an affine_batchnorm_relu() function in layer_utils.py.
    #
    #   DROPOUT: If dropout is non-zero, insert a dropout layer after
    #   every ReLU layer.
    # ============================================================ #
    hidden, hidden_cache = [], []
```

```
    #same as in the init function
    for i in range(1, self.num_layers + 1):
        W = 'W' + str(i)
        b = 'b' + str(i)
        gam = 'gamma' + str(i)
        beta = "beta" + str(i)

        if i == self.num_layers:
            dup = affine_forward(hidden[i-2], self.params[W], self.params[b])
            scores, obj = dup

        else:

            #do hidden first
            if i == 1:
                hidden_param = X
            else:
                hidden_param = hidden[i-2]

            #try to prevent duplicate code
            if self.use_batchnorm :
                appendVar = affine_batchnorm_forward(hidden_param, self.
↪params[W], self.params[b], self.params[gam], self.params[beta], self.
↪bn_params[i-1])
            else:
                appendVar = affine_relu_forward(hidden_param, self.params[W],␣
↪self.params[b])

            hidden.append(appendVar[0])

            #then do hidden_cache
            obj = appendVar[1]

        hidden_cache.append(obj)

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

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

    loss, grads = 0.0, {}
    # ============================================================ #
    # YOUR CODE HERE:
```

```
    #   Implement the backwards pass of the FC net and store the gradients
    #   in the grads dict, so that grads[k] is the gradient of self.params[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    #
    #   BATCHNORM: Incorporate the backward pass of the batchnorm.
    #
    #   DROPOUT: Incorporate the backward pass of dropout.
    # ============================================================ #
    loss, dscores = softmax_loss(scores, y)
    dhidden = [] #update this each iteration

    #go backwards
    for i in range(self.num_layers, 0, -1):
        W = 'W' + str(i)
        b = 'b' + str(i)
        gam = 'gamma' + str(i)
        beta = 'beta' + str(i)

        sum_term = self.params[W] * self.params[W]
        tot_term = 0.5 * self.reg * np.sum(sum_term)

        loss = loss + tot_term

        if i == self.num_layers:
            dh, grads[W], grads[b] = affine_backward(dscores, hidden_cache[self.
↪num_layers-1])
        else:
            if self.use_batchnorm:
                dh, grads[W], grads[b], grads[gam], grads[beta] =␣
↪affine_batchnorm_backward(dh, hidden_cache[i-1])

            else:
                dh, grads[W], grads[b] = affine_relu_backward(dh,␣
↪hidden_cache[i-1])

        dhidden.append(dh)

        grads[W] += self.reg * self.params[W] #need to have tuple unpacking␣
↪above for this to work


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

    return loss, grads
```

1.7.2 layers_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


def affine_relu_backward(dout, cache):
  """
  Backward pass for the affine-relu convenience layer
  """
  fc_cache, relu_cache = cache
  da = relu_backward(dout, relu_cache)
  dx, dw, db = affine_backward(da, fc_cache)
  return dx, dw, db


def affine_batchnorm_forward(x, w, b, gamma, beta, alpha): #batchnorm parameter␣
 ↪-> alpha

    affine_out, affine_cache = affine_forward(x, w, b)
```

```
    batchnorm_out, batchnorm_cache = batchnorm_forward(affine_out, gamma, beta,␣
 ↪alpha)

    y, rcache = relu_forward(batchnorm_out) #y is what we return, also y =␣
 ↪gamma*x + beta

    #rewrite cache
    cache = affine_cache, batchnorm_cache, rcache

    return y, cache

def affine_batchnorm_backward(dout, cache):

    #following the affine_relu_forward and affine_relu_backward examples
    affine_cache, batchnorm_cache, rcache = cache
    da = relu_backward(dout, rcache)

    daff, dgam, dbet = batchnorm_backward(da, batchnorm_cache)
    dx, dw, db = affine_backward(daff, affine_cache)

    return dx, dw, db, dgam, dbet
```

1.7.3 layers.py

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

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

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

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

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

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ============================================================ #
  out = np.dot(x.reshape(x.shape[0], -1), w) + b
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #

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


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

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

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the gradients for the backward pass.
```

```
  # Notice:
  #   dout is N x M
  #   dx should be N x d1 x ... x dk; it relates to dout through multiplication␣
 ↪with w, which is D x M
  #   dw should be D x M; it relates to dout through multiplication with x,␣
 ↪which is N x D after reshaping
  #   db should be M; it is just the sum over dout examples
  # ================================================================ #
  dx = np.dot(dout, w.T).reshape(x.shape)

  db = np.sum(dout, axis=0)

  dw = np.dot(x.reshape(x.shape[0], -1).T, dout)

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

  return dx, dw, db

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

  Input:
  - x: Inputs, of any shape

  Returns a tuple of:
  - out: Output, of the same shape as x
  - cache: x
  """
  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU forward pass.
  # ================================================================ #
  relu = lambda x : x * (x > 0)
  out = relu(x)
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  cache = x
  return out, cache


def relu_backward(dout, cache):
  """
```

```
  Computes the backward pass for a layer of rectified linear units (ReLUs).

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

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ============================================================ #
  # ReLU directs linearly to those > 0
  dx = dout * (x.reshape(x.shape[0], -1) >= 0)

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

  return dx

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

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

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

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

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

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

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

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype)) #key,␣
↪value to be returned if key is not found
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

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

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

      #what is the point of running_mean and running_var?? idk

      #from wk5_coding_dl video

      #step 1 -- calculate sample  mean + variance
      mu = np.mean(x, axis=0)
      sigma_squared = np.var(x, axis=0)
```

```
    #calculate running mean + variance
    running_mean = momentum * running_mean + (1.0 - momentum) * mu
    running_var = momentum * running_var + (1.0 - momentum) * sigma_squared

    #step 2 -- normalize with the *sample* mean and variance (Piazza @368)␣
↪[what does this mean?]
    xhat = (x - mu) / np.sqrt(sigma_squared + eps)

    #step 3 -- scale and shift

    #y -> out
    out = gamma*xhat + beta

    cache = (gamma, beta, running_var, running_mean, mu, sigma_squared, x, eps,␣
↪xhat)
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
  elif mode == 'test':

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

    #Piazza #343 -> only 1 vector, so sample mean -> running mean
    xhat = (x - running_mean) / np.sqrt(running_var + eps)
    out = gamma*xhat + beta


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

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

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

  return out, cache

def batchnorm_backward(dout, cache):
```

```
    """
    Backward pass for batch normalization.

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

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

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

    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
    # ============================================================= #
    gamma, beta, running_var, running_mean, mu, sigma_squared, x, eps, xhat =␣
    ↪cache #tuple unpacking

    N, D = dout.shape #rows, cols

    #from lecture slides: page 47 of lecture9_training, annotated

    dL_dxhat = dout * gamma

    temp1 = sigma_squared + eps

    #calculating dL_dsigma_squared
    first_part = - (x - mu) / (2 * temp1 ** (3/2))
    inside_sum = first_part * dL_dxhat

    #axis=0 means along the column, axis=1 means along the row; returns an array

    #when I got rid of axis=0, my error went from 10^-12 -> 1 lol
    dL_dsigma_squared = np.sum(inside_sum, axis=0)

    #calculated dL_dmu
    dL_dmu = - np.sum(dL_dxhat, axis=0) / np.sqrt(temp1)

    #calculate dL_dx = dx
    #in this case, m = N
```

```
  term1 = dL_dxhat / np.sqrt(temp1)
  term2 = 2 * (x - mu) * dL_dsigma_squared / N
  term3 = dL_dmu / N

  dx = term1 + term2 + term3

  #dbeta and dgamma can be computed from page 44 of lecture9_training slides
  dbeta = np.sum(dout, axis=0)
  dgamma = np.sum(dout*xhat, axis=0) #is axis=0 even needed?

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

  return dx, dgamma, dbeta

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

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

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

  mask = None
  out = None

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

```
    #   dropout mask as the variable mask.
    # ================================================================ #
    pass
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

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

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

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

  return out, cache

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

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

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

```
    #   Implement the inverted dropout backward pass during test time.
    # ================================================================ #
    pass
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
  return dx

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

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

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """
  N = x.shape[0]
  correct_class_scores = x[np.arange(N), y]
  margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
  margins[np.arange(N), y] = 0
  loss = np.sum(margins) / N
  num_pos = np.sum(margins > 0, axis=1)
  dx = np.zeros_like(x)
  dx[margins > 0] = 1
  dx[np.arange(N), y] -= num_pos
  dx /= N
  return loss, dx


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

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

  Returns a tuple of:
  - loss: Scalar giving the loss
```

```
  - dx: Gradient of the loss with respect to x
  """

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

Dropout_Colab

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).

[20]:
```
## 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
```

[21]:
```
# 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,)
```

[22]:
```
!pwd #using Google Colab to figure out the path
```

```
/content/ECE-247-Winter-2021-Kao/hw4/HW4-code
```

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.

[23]:
```
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:  9.997970640146047
Mean of train-time output:  10.004972595239563
Mean of test-time output:  9.997970640146047
Fraction of train-time output set to zero:  0.699716
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.997970640146047
Mean of train-time output:  9.988337318554098
Mean of test-time output:  9.997970640146047
Fraction of train-time output set to zero:  0.40068
Fraction of test-time output set to zero:  0.0
```

```
Running tests with p =  0.75
Mean of input:  9.997970640146047
Mean of train-time output:  9.981088751526062
Mean of test-time output:  9.997970640146047
Fraction of train-time output set to zero:  0.251212
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:

[24]:
```
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:  5.445612783997727e-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).

[25]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [0, 0.25, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

  loss, grads = model.loss(X, y)
```

```
  print('Initial loss: ', loss)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
↪h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num,␣
↪grads[name])))
  print('\n')
```

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


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


Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.553387957138422e-08
W2 relative error: 2.974218050584597e-08
W3 relative error: 4.3413247403122424e-07
b1 relative error: 1.872462967441693e-08
b2 relative error: 5.045591219274328e-09
b3 relative error: 7.487013797161614e-11
```

1.4 Dropout as a regularizer

In class, we claimed that dropout acts as a regularizer by effectively bagging. To check this, we will train two small networks, one with dropout and one without dropout.

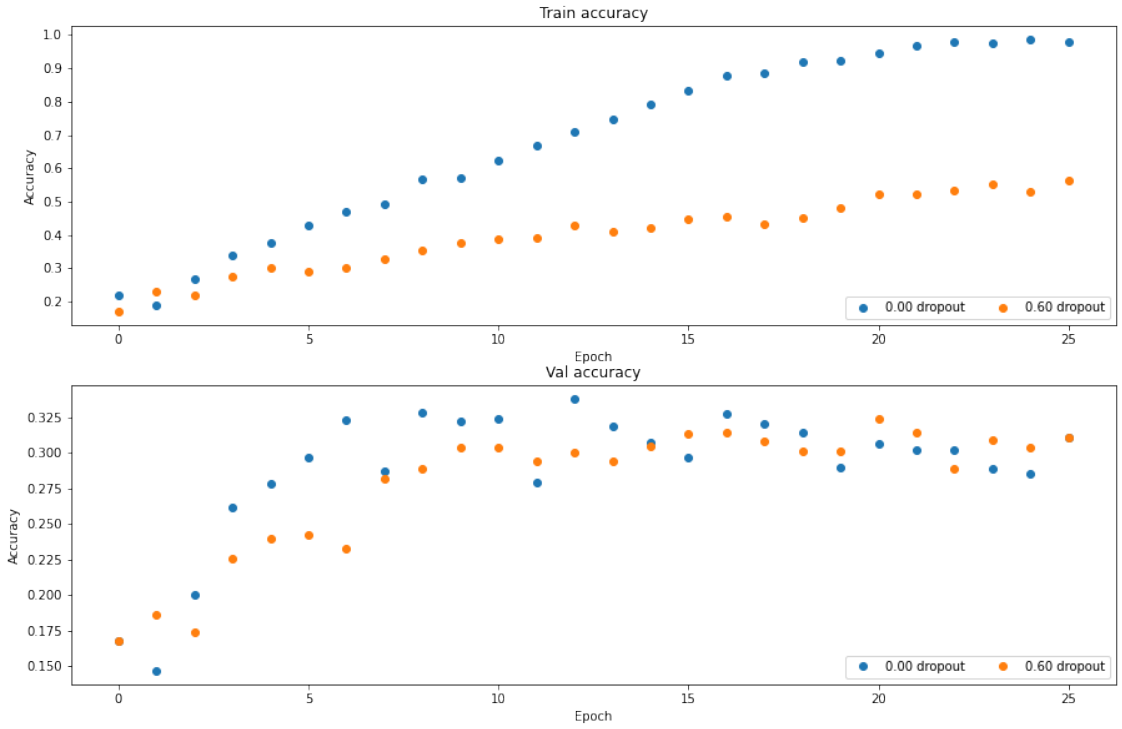
[26]:
```
# Train two identical nets, one with dropout and one without

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: 2.298801
(Epoch 0 / 25) train acc: 0.170000; val_acc: 0.168000
(Epoch 1 / 25) train acc: 0.232000; val_acc: 0.186000
(Epoch 2 / 25) train acc: 0.220000; val_acc: 0.174000
(Epoch 3 / 25) train acc: 0.276000; val_acc: 0.226000
(Epoch 4 / 25) train acc: 0.302000; val_acc: 0.240000
(Epoch 5 / 25) train acc: 0.292000; val_acc: 0.242000
(Epoch 6 / 25) train acc: 0.302000; val_acc: 0.233000
(Epoch 7 / 25) train acc: 0.328000; val_acc: 0.282000
(Epoch 8 / 25) train acc: 0.352000; val_acc: 0.289000
(Epoch 9 / 25) train acc: 0.376000; val_acc: 0.304000
(Epoch 10 / 25) train acc: 0.386000; val_acc: 0.304000
(Epoch 11 / 25) train acc: 0.392000; val_acc: 0.294000
(Epoch 12 / 25) train acc: 0.430000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.410000; val_acc: 0.294000
(Epoch 14 / 25) train acc: 0.422000; val_acc: 0.305000
(Epoch 15 / 25) train acc: 0.448000; val_acc: 0.313000
(Epoch 16 / 25) train acc: 0.456000; val_acc: 0.314000
(Epoch 17 / 25) train acc: 0.434000; val_acc: 0.308000
(Epoch 18 / 25) train acc: 0.450000; val_acc: 0.301000
(Epoch 19 / 25) train acc: 0.482000; val_acc: 0.301000
(Epoch 20 / 25) train acc: 0.524000; val_acc: 0.324000
(Iteration 101 / 125) loss: 1.644595
(Epoch 21 / 25) train acc: 0.522000; val_acc: 0.314000
(Epoch 22 / 25) train acc: 0.532000; val_acc: 0.289000
(Epoch 23 / 25) train acc: 0.552000; val_acc: 0.309000
(Epoch 24 / 25) train acc: 0.530000; val_acc: 0.304000
(Epoch 25 / 25) train acc: 0.564000; val_acc: 0.311000
```

[27]:
```python
# Plot train and validation accuracies of the two models

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

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
```

```
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' %␣
 ↪dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' %␣
 ↪dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

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



1.5 Question

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

1.6 Answer:

One of the goals of dropout is to reduce the sensitivity to the specific weights of the neurons, aka a network with better generalization and a less likely chance of overfitting the training data.

Based on the first graph, we can guess that the baseline model is overfitting the training data, due to its high accuracy; the dropout model, on the other hand is not overfitting the training data, since both have similar accuracies in the validation graph.

In other words, dropout *is* performing regularization, since training_accuracy - validation_accuracy is smaller for the dropout NN, than the baseline NN.

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.

1.7.1 First Attempt

For the first attempt, I tried running the following code on my laptop. This really overtaxed my laptop and took a really long time.

[28]:
```
%%time

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

#just guess and check the parameters
layer_dims = [500, 500, 500, 500]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.5, use_batchnorm=True)

#solver

#default lr from optim.py notebook
lr = 1e-2

#default lr_decay from optim.py notebook
lr_decay = 0.99

#based on the Optimization notebook, we know that 'adam' is the best␣
 ↪update_rule
```

```
solver = Solver(nn, data, num_epochs = 10, batch_size = 100, update_rule =␣
 ↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 25)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 4900) loss: 2.332860
(Epoch 0 / 10) train acc: 0.116000; val_acc: 0.105000
(Iteration 26 / 4900) loss: 2.199181
(Iteration 51 / 4900) loss: 2.189094
(Iteration 76 / 4900) loss: 2.191130
(Iteration 101 / 4900) loss: 2.140513
(Iteration 126 / 4900) loss: 1.864191
(Iteration 151 / 4900) loss: 1.826431
(Iteration 176 / 4900) loss: 2.189512
(Iteration 201 / 4900) loss: 1.592877
(Iteration 226 / 4900) loss: 1.921537
(Iteration 251 / 4900) loss: 2.108006
(Iteration 276 / 4900) loss: 2.132851
(Iteration 301 / 4900) loss: 1.850874
(Iteration 326 / 4900) loss: 1.888745
(Iteration 351 / 4900) loss: 1.930197
(Iteration 376 / 4900) loss: 1.826653
(Iteration 401 / 4900) loss: 1.690391
(Iteration 426 / 4900) loss: 1.868751
(Iteration 451 / 4900) loss: 1.891438
(Iteration 476 / 4900) loss: 1.701267
(Epoch 1 / 10) train acc: 0.382000; val_acc: 0.418000
(Iteration 501 / 4900) loss: 1.809092
(Iteration 526 / 4900) loss: 1.697924
(Iteration 551 / 4900) loss: 1.850683
(Iteration 576 / 4900) loss: 1.692670
(Iteration 601 / 4900) loss: 2.030057
(Iteration 626 / 4900) loss: 1.769373
(Iteration 651 / 4900) loss: 1.785969
(Iteration 676 / 4900) loss: 1.718198
(Iteration 701 / 4900) loss: 1.925041
(Iteration 726 / 4900) loss: 1.854148
```

```
(Iteration 751 / 4900) loss: 1.732306
(Iteration 776 / 4900) loss: 1.809648
(Iteration 801 / 4900) loss: 1.754564
(Iteration 826 / 4900) loss: 1.767626
(Iteration 851 / 4900) loss: 1.686986
(Iteration 876 / 4900) loss: 1.867530
(Iteration 901 / 4900) loss: 1.686788
(Iteration 926 / 4900) loss: 1.705472
(Iteration 951 / 4900) loss: 1.697577
(Iteration 976 / 4900) loss: 1.776239
(Epoch 2 / 10) train acc: 0.462000; val_acc: 0.461000
(Iteration 1001 / 4900) loss: 1.694596
(Iteration 1026 / 4900) loss: 1.719853
(Iteration 1051 / 4900) loss: 1.672360
(Iteration 1076 / 4900) loss: 1.902215
(Iteration 1101 / 4900) loss: 1.890265
(Iteration 1126 / 4900) loss: 1.989341
(Iteration 1151 / 4900) loss: 1.660316
(Iteration 1176 / 4900) loss: 1.843407
(Iteration 1201 / 4900) loss: 1.874394
(Iteration 1226 / 4900) loss: 1.881918
(Iteration 1251 / 4900) loss: 1.639835
(Iteration 1276 / 4900) loss: 1.824072
(Iteration 1301 / 4900) loss: 1.551497
(Iteration 1326 / 4900) loss: 1.887207
(Iteration 1351 / 4900) loss: 1.481773
(Iteration 1376 / 4900) loss: 1.661601
(Iteration 1401 / 4900) loss: 1.473951
(Iteration 1426 / 4900) loss: 1.737493
(Iteration 1451 / 4900) loss: 1.679041
(Epoch 3 / 10) train acc: 0.479000; val_acc: 0.470000
(Iteration 1476 / 4900) loss: 1.602146
(Iteration 1501 / 4900) loss: 1.748989
(Iteration 1526 / 4900) loss: 1.609776
(Iteration 1551 / 4900) loss: 1.650858
(Iteration 1576 / 4900) loss: 1.519845
(Iteration 1601 / 4900) loss: 1.721005
(Iteration 1626 / 4900) loss: 1.670260
(Iteration 1651 / 4900) loss: 1.665284
(Iteration 1676 / 4900) loss: 1.627090
(Iteration 1701 / 4900) loss: 1.729870
(Iteration 1726 / 4900) loss: 1.617798
(Iteration 1751 / 4900) loss: 1.879724
(Iteration 1776 / 4900) loss: 1.712234
(Iteration 1801 / 4900) loss: 1.711639
(Iteration 1826 / 4900) loss: 1.667704
(Iteration 1851 / 4900) loss: 1.751753
(Iteration 1876 / 4900) loss: 1.625569
```

```
(Iteration 1901 / 4900) loss: 1.631837
(Iteration 1926 / 4900) loss: 1.641124
(Iteration 1951 / 4900) loss: 1.561766
(Epoch 4 / 10) train acc: 0.491000; val_acc: 0.477000
(Iteration 1976 / 4900) loss: 1.439547
(Iteration 2001 / 4900) loss: 1.630475
(Iteration 2026 / 4900) loss: 1.845729
(Iteration 2051 / 4900) loss: 1.678888
(Iteration 2076 / 4900) loss: 1.710085
(Iteration 2101 / 4900) loss: 1.608200
(Iteration 2126 / 4900) loss: 1.531925
(Iteration 2151 / 4900) loss: 1.463512
(Iteration 2176 / 4900) loss: 1.678392
(Iteration 2201 / 4900) loss: 1.567090
(Iteration 2226 / 4900) loss: 1.543012
(Iteration 2251 / 4900) loss: 1.454875
(Iteration 2276 / 4900) loss: 1.643265
(Iteration 2301 / 4900) loss: 1.871612
(Iteration 2326 / 4900) loss: 1.508889
(Iteration 2351 / 4900) loss: 1.624331
(Iteration 2376 / 4900) loss: 1.372808
(Iteration 2401 / 4900) loss: 1.562961
(Iteration 2426 / 4900) loss: 1.457379
(Epoch 5 / 10) train acc: 0.506000; val_acc: 0.478000
(Iteration 2451 / 4900) loss: 1.548133
(Iteration 2476 / 4900) loss: 1.640547
(Iteration 2501 / 4900) loss: 1.490865
(Iteration 2526 / 4900) loss: 1.553773
(Iteration 2551 / 4900) loss: 1.437294
(Iteration 2576 / 4900) loss: 1.527256
(Iteration 2601 / 4900) loss: 1.416247
(Iteration 2626 / 4900) loss: 1.717501
(Iteration 2651 / 4900) loss: 1.500852
(Iteration 2676 / 4900) loss: 1.670706
(Iteration 2701 / 4900) loss: 1.721263
(Iteration 2726 / 4900) loss: 1.592374
(Iteration 2751 / 4900) loss: 1.696896
(Iteration 2776 / 4900) loss: 1.295017
(Iteration 2801 / 4900) loss: 1.630686
(Iteration 2826 / 4900) loss: 1.537989
(Iteration 2851 / 4900) loss: 1.523463
(Iteration 2876 / 4900) loss: 1.712791
(Iteration 2901 / 4900) loss: 1.735747
(Iteration 2926 / 4900) loss: 1.584328
(Epoch 6 / 10) train acc: 0.511000; val_acc: 0.501000
(Iteration 2951 / 4900) loss: 1.708704
(Iteration 2976 / 4900) loss: 1.674250
(Iteration 3001 / 4900) loss: 1.371346
```

```
(Iteration 3026 / 4900) loss: 1.493894
(Iteration 3051 / 4900) loss: 1.536961
(Iteration 3076 / 4900) loss: 1.885181
(Iteration 3101 / 4900) loss: 1.545976
(Iteration 3126 / 4900) loss: 1.431801
(Iteration 3151 / 4900) loss: 1.506173
(Iteration 3176 / 4900) loss: 1.523720
(Iteration 3201 / 4900) loss: 1.464956
(Iteration 3226 / 4900) loss: 1.778396
(Iteration 3251 / 4900) loss: 1.748724
(Iteration 3276 / 4900) loss: 1.326606
(Iteration 3301 / 4900) loss: 1.484228
(Iteration 3326 / 4900) loss: 1.510274
(Iteration 3351 / 4900) loss: 1.448585
(Iteration 3376 / 4900) loss: 1.740505
(Iteration 3401 / 4900) loss: 1.697072
(Iteration 3426 / 4900) loss: 1.687337
(Epoch 7 / 10) train acc: 0.498000; val_acc: 0.500000
(Iteration 3451 / 4900) loss: 1.701744
(Iteration 3476 / 4900) loss: 1.602375
(Iteration 3501 / 4900) loss: 1.648847
(Iteration 3526 / 4900) loss: 1.757738
(Iteration 3551 / 4900) loss: 1.555254
(Iteration 3576 / 4900) loss: 1.492249
(Iteration 3601 / 4900) loss: 1.593187
(Iteration 3626 / 4900) loss: 1.508492
(Iteration 3651 / 4900) loss: 1.475083
(Iteration 3676 / 4900) loss: 1.419426
(Iteration 3701 / 4900) loss: 1.480140
(Iteration 3726 / 4900) loss: 1.454613
(Iteration 3751 / 4900) loss: 1.524612
(Iteration 3776 / 4900) loss: 1.798707
(Iteration 3801 / 4900) loss: 1.342462
(Iteration 3826 / 4900) loss: 1.515313
(Iteration 3851 / 4900) loss: 1.421899
(Iteration 3876 / 4900) loss: 1.675039
(Iteration 3901 / 4900) loss: 1.667227
(Epoch 8 / 10) train acc: 0.520000; val_acc: 0.504000
(Iteration 3926 / 4900) loss: 1.477817
(Iteration 3951 / 4900) loss: 1.595445
(Iteration 3976 / 4900) loss: 1.507753
(Iteration 4001 / 4900) loss: 1.733239
(Iteration 4026 / 4900) loss: 1.511092
(Iteration 4051 / 4900) loss: 1.525700
(Iteration 4076 / 4900) loss: 1.729488
(Iteration 4101 / 4900) loss: 1.459962
(Iteration 4126 / 4900) loss: 1.705323
(Iteration 4151 / 4900) loss: 1.549018
```

```
(Iteration 4176 / 4900) loss: 1.545604
(Iteration 4201 / 4900) loss: 1.600078
(Iteration 4226 / 4900) loss: 1.572480
(Iteration 4251 / 4900) loss: 1.497264
(Iteration 4276 / 4900) loss: 1.481055
(Iteration 4301 / 4900) loss: 1.504818
(Iteration 4326 / 4900) loss: 1.466714
(Iteration 4351 / 4900) loss: 1.533869
(Iteration 4376 / 4900) loss: 1.414315
(Iteration 4401 / 4900) loss: 1.449522
(Epoch 9 / 10) train acc: 0.523000; val_acc: 0.529000
(Iteration 4426 / 4900) loss: 1.445199
(Iteration 4451 / 4900) loss: 1.356018
(Iteration 4476 / 4900) loss: 1.281951
(Iteration 4501 / 4900) loss: 1.479207
(Iteration 4526 / 4900) loss: 1.514100
(Iteration 4551 / 4900) loss: 1.399769
(Iteration 4576 / 4900) loss: 1.327503
(Iteration 4601 / 4900) loss: 1.587030
(Iteration 4626 / 4900) loss: 1.557214
(Iteration 4651 / 4900) loss: 1.517289
(Iteration 4676 / 4900) loss: 1.588984
(Iteration 4701 / 4900) loss: 1.478592
(Iteration 4726 / 4900) loss: 1.565103
(Iteration 4751 / 4900) loss: 1.374198
(Iteration 4776 / 4900) loss: 1.447602
(Iteration 4801 / 4900) loss: 1.739847
(Iteration 4826 / 4900) loss: 1.536477
(Iteration 4851 / 4900) loss: 1.511670
(Iteration 4876 / 4900) loss: 1.618257
(Epoch 10 / 10) train acc: 0.511000; val_acc: 0.531000
CPU times: user 15min 42s, sys: 7min 28s, total: 23min 11s
Wall time: 11min 50s
```

1.7.2 Second Attempt:

The first attempt took 54 minutes and wasn't even close to the 60 percent validation accuracy needed. Next, I changed the dropout parameter, learning rate, and learning rate decay, and got closer, but it still took 40 minutes, so I decided to use Google Colab for the last part of the project.

[29]:
```
%%time

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

```
#just guess and check the parameters
layer_dims = [500, 500, 500, 500]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.75, use_batchnorm=True)

#solver

#default lr from optim.py notebook
lr = 1e-3

#default lr_decay from optim.py notebook
lr_decay = 0.95

#based on the Optimization notebook, we know that 'adam' is the best␣
 ↪update_rule

solver = Solver(nn, data, num_epochs = 10, batch_size = 100, update_rule =␣
 ↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 100)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 4900) loss: 2.352284
(Epoch 0 / 10) train acc: 0.193000; val_acc: 0.187000
(Iteration 101 / 4900) loss: 1.855497
(Iteration 201 / 4900) loss: 1.540748
(Iteration 301 / 4900) loss: 1.645662
(Iteration 401 / 4900) loss: 1.495835
(Epoch 1 / 10) train acc: 0.464000; val_acc: 0.468000
(Iteration 501 / 4900) loss: 1.626935
(Iteration 601 / 4900) loss: 1.285022
(Iteration 701 / 4900) loss: 1.707425
(Iteration 801 / 4900) loss: 1.508795
(Iteration 901 / 4900) loss: 1.513536
(Epoch 2 / 10) train acc: 0.495000; val_acc: 0.498000
(Iteration 1001 / 4900) loss: 1.253548
(Iteration 1101 / 4900) loss: 1.430601
```

```
(Iteration 1201 / 4900) loss: 1.199624
(Iteration 1301 / 4900) loss: 1.463422
(Iteration 1401 / 4900) loss: 1.454099
(Epoch 3 / 10) train acc: 0.566000; val_acc: 0.514000
(Iteration 1501 / 4900) loss: 1.341385
(Iteration 1601 / 4900) loss: 1.407252
(Iteration 1701 / 4900) loss: 1.393968
(Iteration 1801 / 4900) loss: 1.320074
(Iteration 1901 / 4900) loss: 1.406331
(Epoch 4 / 10) train acc: 0.567000; val_acc: 0.522000
(Iteration 2001 / 4900) loss: 1.214926
(Iteration 2101 / 4900) loss: 1.409119
(Iteration 2201 / 4900) loss: 1.349190
(Iteration 2301 / 4900) loss: 1.320982
(Iteration 2401 / 4900) loss: 1.257009
(Epoch 5 / 10) train acc: 0.572000; val_acc: 0.531000
(Iteration 2501 / 4900) loss: 1.188343
(Iteration 2601 / 4900) loss: 1.261243
(Iteration 2701 / 4900) loss: 1.127422
(Iteration 2801 / 4900) loss: 1.296153
(Iteration 2901 / 4900) loss: 1.345685
(Epoch 6 / 10) train acc: 0.606000; val_acc: 0.546000
(Iteration 3001 / 4900) loss: 1.028016
(Iteration 3101 / 4900) loss: 1.186314
(Iteration 3201 / 4900) loss: 1.084921
(Iteration 3301 / 4900) loss: 1.254569
(Iteration 3401 / 4900) loss: 1.315355
(Epoch 7 / 10) train acc: 0.613000; val_acc: 0.555000
(Iteration 3501 / 4900) loss: 1.299533
(Iteration 3601 / 4900) loss: 1.388772
(Iteration 3701 / 4900) loss: 1.275999
(Iteration 3801 / 4900) loss: 1.333248
(Iteration 3901 / 4900) loss: 1.154237
(Epoch 8 / 10) train acc: 0.638000; val_acc: 0.562000
(Iteration 4001 / 4900) loss: 1.165062
(Iteration 4101 / 4900) loss: 1.364126
(Iteration 4201 / 4900) loss: 1.416606
(Iteration 4301 / 4900) loss: 1.157408
(Iteration 4401 / 4900) loss: 1.238314
(Epoch 9 / 10) train acc: 0.648000; val_acc: 0.572000
(Iteration 4501 / 4900) loss: 1.160329
(Iteration 4601 / 4900) loss: 1.445909
(Iteration 4701 / 4900) loss: 1.106907
(Iteration 4801 / 4900) loss: 1.069607
(Epoch 10 / 10) train acc: 0.666000; val_acc: 0.558000
CPU times: user 15min 40s, sys: 7min 26s, total: 23min 7s
Wall time: 11min 46s
```

1.7.3 Attempt 3:

As mentioned above, in this attempt I used Google Colab to speed up training time.

[30]:
```
%%time

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


#just guess and check the parameters
layer_dims = [350, 350, 350, 350]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.85, use_batchnorm=True)

#solver

#default lr from optim.py notebook
lr = 1e-3

#default lr_decay from optim.py notebook
lr_decay = 0.95

#based on the Optimization notebook, we know that 'adam' is the best␣
↪update_rule

solver = Solver(nn, data, num_epochs = 10, batch_size = 100, update_rule =␣
↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 100)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 4900) loss: 2.295683
(Epoch 0 / 10) train acc: 0.179000; val_acc: 0.204000
(Iteration 101 / 4900) loss: 1.818607
```

```
(Iteration 201 / 4900) loss: 1.553953
(Iteration 301 / 4900) loss: 1.520726
(Iteration 401 / 4900) loss: 1.490148
(Epoch 1 / 10) train acc: 0.482000; val_acc: 0.449000
(Iteration 501 / 4900) loss: 1.544304
(Iteration 601 / 4900) loss: 1.258895
(Iteration 701 / 4900) loss: 1.398603
(Iteration 801 / 4900) loss: 1.499660
(Iteration 901 / 4900) loss: 1.516559
(Epoch 2 / 10) train acc: 0.528000; val_acc: 0.525000
(Iteration 1001 / 4900) loss: 1.316359
(Iteration 1101 / 4900) loss: 1.345274
(Iteration 1201 / 4900) loss: 1.079603
(Iteration 1301 / 4900) loss: 1.442688
(Iteration 1401 / 4900) loss: 1.351874
(Epoch 3 / 10) train acc: 0.538000; val_acc: 0.515000
(Iteration 1501 / 4900) loss: 1.398941
(Iteration 1601 / 4900) loss: 1.308362
(Iteration 1701 / 4900) loss: 1.216513
(Iteration 1801 / 4900) loss: 1.276661
(Iteration 1901 / 4900) loss: 1.273022
(Epoch 4 / 10) train acc: 0.573000; val_acc: 0.544000
(Iteration 2001 / 4900) loss: 1.286439
(Iteration 2101 / 4900) loss: 1.203449
(Iteration 2201 / 4900) loss: 1.122837
(Iteration 2301 / 4900) loss: 1.225562
(Iteration 2401 / 4900) loss: 1.035807
(Epoch 5 / 10) train acc: 0.591000; val_acc: 0.547000
(Iteration 2501 / 4900) loss: 1.189923
(Iteration 2601 / 4900) loss: 1.226721
(Iteration 2701 / 4900) loss: 1.082709
(Iteration 2801 / 4900) loss: 1.197990
(Iteration 2901 / 4900) loss: 1.271250
(Epoch 6 / 10) train acc: 0.610000; val_acc: 0.557000
(Iteration 3001 / 4900) loss: 1.060765
(Iteration 3101 / 4900) loss: 1.090462
(Iteration 3201 / 4900) loss: 1.295801
(Iteration 3301 / 4900) loss: 1.196382
(Iteration 3401 / 4900) loss: 1.044863
(Epoch 7 / 10) train acc: 0.621000; val_acc: 0.554000
(Iteration 3501 / 4900) loss: 1.108505
(Iteration 3601 / 4900) loss: 1.271705
(Iteration 3701 / 4900) loss: 1.018351
(Iteration 3801 / 4900) loss: 1.333046
(Iteration 3901 / 4900) loss: 1.340748
(Epoch 8 / 10) train acc: 0.650000; val_acc: 0.556000
(Iteration 4001 / 4900) loss: 1.158957
(Iteration 4101 / 4900) loss: 0.974469
```

```
(Iteration 4201 / 4900) loss: 1.096994
(Iteration 4301 / 4900) loss: 1.187967
(Iteration 4401 / 4900) loss: 1.036646
(Epoch 9 / 10) train acc: 0.669000; val_acc: 0.552000
(Iteration 4501 / 4900) loss: 1.239225
(Iteration 4601 / 4900) loss: 0.992910
(Iteration 4701 / 4900) loss: 0.912345
(Iteration 4801 / 4900) loss: 0.914731
(Epoch 10 / 10) train acc: 0.643000; val_acc: 0.575000
CPU times: user 10min 30s, sys: 5min 20s, total: 15min 50s
Wall time: 8min
```

1.7.4 Attempt 4:

Switching to colab changed the training from 40 minutes to around 10 minutes, which is a lot faster, but still a long time. The above attempt seems promising, but it looks like iterations need to be increased, since it doesn't look like the gradient converged. So I'll run again with more epochs. Additionally, I'll try changing the weight_scale parameter from its default value (0.01), to maybe force an earlier converge.

[32]:
```
%%time

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


#just guess and check the parameters
layer_dims = [350, 350, 350, 350]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.85, use_batchnorm=True,␣
 ↪weight_scale = 0.03)

#solver

#default lr from optim.py notebook
lr = 1e-3

#default lr_decay from optim.py notebook
lr_decay = 0.95

#based on the Optimization notebook, we know that 'adam' is the best␣
 ↪update_rule
```

```
solver = Solver(nn, data, num_epochs = 20, batch_size = 100, update_rule =␣
 ↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 100)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 9800) loss: 2.426864
(Epoch 0 / 20) train acc: 0.110000; val_acc: 0.127000
(Iteration 101 / 9800) loss: 1.716629
(Iteration 201 / 9800) loss: 1.621298
(Iteration 301 / 9800) loss: 1.595061
(Iteration 401 / 9800) loss: 1.562907
(Epoch 1 / 20) train acc: 0.489000; val_acc: 0.470000
(Iteration 501 / 9800) loss: 1.471968
(Iteration 601 / 9800) loss: 1.639235
(Iteration 701 / 9800) loss: 1.635399
(Iteration 801 / 9800) loss: 1.350104
(Iteration 901 / 9800) loss: 1.366536
(Epoch 2 / 20) train acc: 0.521000; val_acc: 0.515000
(Iteration 1001 / 9800) loss: 1.444865
(Iteration 1101 / 9800) loss: 1.341671
(Iteration 1201 / 9800) loss: 1.369643
(Iteration 1301 / 9800) loss: 1.111501
(Iteration 1401 / 9800) loss: 1.353666
(Epoch 3 / 20) train acc: 0.563000; val_acc: 0.538000
(Iteration 1501 / 9800) loss: 1.219532
(Iteration 1601 / 9800) loss: 1.491365
(Iteration 1701 / 9800) loss: 1.328477
(Iteration 1801 / 9800) loss: 1.436202
(Iteration 1901 / 9800) loss: 1.116000
(Epoch 4 / 20) train acc: 0.591000; val_acc: 0.533000
(Iteration 2001 / 9800) loss: 1.226051
(Iteration 2101 / 9800) loss: 1.013112
(Iteration 2201 / 9800) loss: 1.354813
(Iteration 2301 / 9800) loss: 1.188410
(Iteration 2401 / 9800) loss: 1.238352
(Epoch 5 / 20) train acc: 0.609000; val_acc: 0.539000
(Iteration 2501 / 9800) loss: 1.088475
(Iteration 2601 / 9800) loss: 1.143734
```

```
(Iteration 2701 / 9800) loss: 1.088841
(Iteration 2801 / 9800) loss: 1.328313
(Iteration 2901 / 9800) loss: 1.090666
(Epoch 6 / 20) train acc: 0.619000; val_acc: 0.522000
(Iteration 3001 / 9800) loss: 1.273766
(Iteration 3101 / 9800) loss: 1.078481
(Iteration 3201 / 9800) loss: 1.075816
(Iteration 3301 / 9800) loss: 0.975710
(Iteration 3401 / 9800) loss: 1.181645
(Epoch 7 / 20) train acc: 0.625000; val_acc: 0.531000
(Iteration 3501 / 9800) loss: 1.167905
(Iteration 3601 / 9800) loss: 1.140606
(Iteration 3701 / 9800) loss: 1.164467
(Iteration 3801 / 9800) loss: 1.258477
(Iteration 3901 / 9800) loss: 1.126213
(Epoch 8 / 20) train acc: 0.624000; val_acc: 0.547000
(Iteration 4001 / 9800) loss: 1.050596
(Iteration 4101 / 9800) loss: 1.032181
(Iteration 4201 / 9800) loss: 0.983441
(Iteration 4301 / 9800) loss: 1.130443
(Iteration 4401 / 9800) loss: 0.927212
(Epoch 9 / 20) train acc: 0.684000; val_acc: 0.547000
(Iteration 4501 / 9800) loss: 0.945351
(Iteration 4601 / 9800) loss: 1.192936
(Iteration 4701 / 9800) loss: 0.949690
(Iteration 4801 / 9800) loss: 1.087105
(Epoch 10 / 20) train acc: 0.676000; val_acc: 0.556000
(Iteration 4901 / 9800) loss: 1.171546
(Iteration 5001 / 9800) loss: 1.064518
(Iteration 5101 / 9800) loss: 1.188300
(Iteration 5201 / 9800) loss: 0.822895
(Iteration 5301 / 9800) loss: 1.132019
(Epoch 11 / 20) train acc: 0.684000; val_acc: 0.572000
(Iteration 5401 / 9800) loss: 0.805991
(Iteration 5501 / 9800) loss: 1.127349
(Iteration 5601 / 9800) loss: 0.945076
(Iteration 5701 / 9800) loss: 1.089295
(Iteration 5801 / 9800) loss: 1.036892
(Epoch 12 / 20) train acc: 0.713000; val_acc: 0.559000
(Iteration 5901 / 9800) loss: 0.887856
(Iteration 6001 / 9800) loss: 0.876571
(Iteration 6101 / 9800) loss: 0.895836
(Iteration 6201 / 9800) loss: 0.969919
(Iteration 6301 / 9800) loss: 0.850819
(Epoch 13 / 20) train acc: 0.727000; val_acc: 0.544000
(Iteration 6401 / 9800) loss: 1.000222
(Iteration 6501 / 9800) loss: 0.970844
(Iteration 6601 / 9800) loss: 0.851073
```

```
(Iteration 6701 / 9800) loss: 0.961441
(Iteration 6801 / 9800) loss: 1.275310
(Epoch 14 / 20) train acc: 0.712000; val_acc: 0.545000
(Iteration 6901 / 9800) loss: 1.152739
(Iteration 7001 / 9800) loss: 0.889409
(Iteration 7101 / 9800) loss: 0.931267
(Iteration 7201 / 9800) loss: 0.952744
(Iteration 7301 / 9800) loss: 0.897852
(Epoch 15 / 20) train acc: 0.745000; val_acc: 0.582000
(Iteration 7401 / 9800) loss: 0.932454
(Iteration 7501 / 9800) loss: 0.886274
(Iteration 7601 / 9800) loss: 0.864786
(Iteration 7701 / 9800) loss: 0.877694
(Iteration 7801 / 9800) loss: 0.862586
(Epoch 16 / 20) train acc: 0.733000; val_acc: 0.573000
(Iteration 7901 / 9800) loss: 0.944406
(Iteration 8001 / 9800) loss: 0.849601
(Iteration 8101 / 9800) loss: 0.934373
(Iteration 8201 / 9800) loss: 1.064792
(Iteration 8301 / 9800) loss: 0.846876
(Epoch 17 / 20) train acc: 0.743000; val_acc: 0.572000
(Iteration 8401 / 9800) loss: 0.788819
(Iteration 8501 / 9800) loss: 0.701147
(Iteration 8601 / 9800) loss: 0.988433
(Iteration 8701 / 9800) loss: 0.821340
(Iteration 8801 / 9800) loss: 0.862189
(Epoch 18 / 20) train acc: 0.768000; val_acc: 0.570000
(Iteration 8901 / 9800) loss: 0.716706
(Iteration 9001 / 9800) loss: 0.697144
(Iteration 9101 / 9800) loss: 0.928793
(Iteration 9201 / 9800) loss: 1.014345
(Iteration 9301 / 9800) loss: 0.871679
(Epoch 19 / 20) train acc: 0.778000; val_acc: 0.567000
(Iteration 9401 / 9800) loss: 0.765553
(Iteration 9501 / 9800) loss: 0.824246
(Iteration 9601 / 9800) loss: 0.660961
(Iteration 9701 / 9800) loss: 0.746656
(Epoch 20 / 20) train acc: 0.783000; val_acc: 0.563000
CPU times: user 20min 52s, sys: 10min 39s, total: 31min 31s
Wall time: 15min 56s
```

1.7.5 Attempt 5:

Looks like our validation accuracy is still below the 60% threshold, but only barely. At this point, let's just try brute forcing, by increasing the batch_size and layer_dims.

[33]:
```
%%time
```

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


#just guess and check the parameters
layer_dims = [500, 500, 500, 500]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.85, use_batchnorm=True,␣
 ↪weight_scale = 0.03)

#solver

#default lr from optim.py notebook
lr = 1e-3

#default lr_decay from optim.py notebook
lr_decay = 0.95

#based on the Optimization notebook, we know that 'adam' is the best␣
 ↪update_rule

solver = Solver(nn, data, num_epochs = 20, batch_size = 500, update_rule =␣
 ↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 100)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 1960) loss: 2.437022
(Epoch 0 / 20) train acc: 0.238000; val_acc: 0.237000
(Epoch 1 / 20) train acc: 0.450000; val_acc: 0.470000
(Iteration 101 / 1960) loss: 1.535537
(Epoch 2 / 20) train acc: 0.545000; val_acc: 0.507000
(Iteration 201 / 1960) loss: 1.395684
(Epoch 3 / 20) train acc: 0.556000; val_acc: 0.515000
```

```
(Iteration 301 / 1960) loss: 1.230348
(Epoch 4 / 20) train acc: 0.601000; val_acc: 0.534000
(Iteration 401 / 1960) loss: 1.251369
(Epoch 5 / 20) train acc: 0.605000; val_acc: 0.557000
(Iteration 501 / 1960) loss: 1.210251
(Epoch 6 / 20) train acc: 0.631000; val_acc: 0.570000
(Iteration 601 / 1960) loss: 1.056810
(Epoch 7 / 20) train acc: 0.668000; val_acc: 0.564000
(Iteration 701 / 1960) loss: 1.080420
(Epoch 8 / 20) train acc: 0.688000; val_acc: 0.566000
(Iteration 801 / 1960) loss: 0.999389
(Epoch 9 / 20) train acc: 0.706000; val_acc: 0.566000
(Iteration 901 / 1960) loss: 0.967713
(Epoch 10 / 20) train acc: 0.712000; val_acc: 0.571000
(Iteration 1001 / 1960) loss: 0.978605
(Epoch 11 / 20) train acc: 0.725000; val_acc: 0.571000
(Iteration 1101 / 1960) loss: 1.035527
(Epoch 12 / 20) train acc: 0.739000; val_acc: 0.578000
(Iteration 1201 / 1960) loss: 0.883730
(Epoch 13 / 20) train acc: 0.750000; val_acc: 0.566000
(Iteration 1301 / 1960) loss: 0.760109
(Epoch 14 / 20) train acc: 0.786000; val_acc: 0.573000
(Iteration 1401 / 1960) loss: 0.820489
(Epoch 15 / 20) train acc: 0.784000; val_acc: 0.571000
(Iteration 1501 / 1960) loss: 0.794237
(Epoch 16 / 20) train acc: 0.805000; val_acc: 0.563000
(Iteration 1601 / 1960) loss: 0.660435
(Epoch 17 / 20) train acc: 0.810000; val_acc: 0.561000
(Iteration 1701 / 1960) loss: 0.784691
(Epoch 18 / 20) train acc: 0.803000; val_acc: 0.561000
(Iteration 1801 / 1960) loss: 0.700675
(Epoch 19 / 20) train acc: 0.846000; val_acc: 0.575000
(Iteration 1901 / 1960) loss: 0.648638
(Epoch 20 / 20) train acc: 0.832000; val_acc: 0.573000
CPU times: user 20min 17s, sys: 5min 33s, total: 25min 51s
Wall time: 13min 3s
```

1.7.6 Attempt 6:

Dang, it looks like we're still stuck. Let's try increasing our weight scale and learning rate, and since the dimensions don't seem to make a difference, we can decrease those for faster runtime. If guess and check doesn't work, I'll probably attempt to do random search or grid search to find the best hyperparameters.

[35]:
```
%%time

# ================================================================ #
# YOUR CODE HERE:
```

```
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #


#just guess and check the parameters
layer_dims = [500, 500, 500, 500]

#use dropout + batchnorm  to prevent overfitting
#keep the other params default
nn = FullyConnectedNet(layer_dims, dropout=0.8, use_batchnorm=True,␣
 ↪weight_scale = 0.05)

#solver

#default lr from optim.py notebook
lr = 1




\e-3

#default lr_decay from optim.py notebook
lr_decay = 0.95

#based on the Optimization notebook, we know that 'adam' is the best␣
 ↪update_rule

solver = Solver(nn, data, num_epochs = 50, batch_size = 100, update_rule =␣
 ↪'adam',
                optim_config = {'learning_rate': lr},
                 lr_decay = lr_decay, verbose=True, print_every = 500)

solver.train()

#seeking to get 60% accuracy


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

```
(Iteration 1 / 24500) loss: 2.505333
(Epoch 0 / 50) train acc: 0.166000; val_acc: 0.140000
(Epoch 1 / 50) train acc: 0.466000; val_acc: 0.481000
```

```
(Iteration 501 / 24500) loss: 1.627242
(Epoch 2 / 50) train acc: 0.541000; val_acc: 0.507000
(Iteration 1001 / 24500) loss: 1.474940
(Epoch 3 / 50) train acc: 0.543000; val_acc: 0.512000
(Iteration 1501 / 24500) loss: 1.243632
(Epoch 4 / 50) train acc: 0.578000; val_acc: 0.532000
(Iteration 2001 / 24500) loss: 1.302482
(Epoch 5 / 50) train acc: 0.611000; val_acc: 0.526000
(Iteration 2501 / 24500) loss: 1.245727
(Epoch 6 / 50) train acc: 0.595000; val_acc: 0.551000
(Iteration 3001 / 24500) loss: 1.106715
(Epoch 7 / 50) train acc: 0.631000; val_acc: 0.546000
(Iteration 3501 / 24500) loss: 1.154325
(Epoch 8 / 50) train acc: 0.641000; val_acc: 0.558000
(Iteration 4001 / 24500) loss: 1.076638
(Epoch 9 / 50) train acc: 0.664000; val_acc: 0.549000
(Iteration 4501 / 24500) loss: 1.153488
(Epoch 10 / 50) train acc: 0.655000; val_acc: 0.546000
(Iteration 5001 / 24500) loss: 0.859077
(Epoch 11 / 50) train acc: 0.687000; val_acc: 0.552000
(Iteration 5501 / 24500) loss: 0.994879
(Epoch 12 / 50) train acc: 0.721000; val_acc: 0.559000
(Iteration 6001 / 24500) loss: 1.060874
(Epoch 13 / 50) train acc: 0.703000; val_acc: 0.562000
(Iteration 6501 / 24500) loss: 0.709178
(Epoch 14 / 50) train acc: 0.707000; val_acc: 0.571000
(Iteration 7001 / 24500) loss: 0.981845
(Epoch 15 / 50) train acc: 0.702000; val_acc: 0.568000
(Iteration 7501 / 24500) loss: 0.920624
(Epoch 16 / 50) train acc: 0.751000; val_acc: 0.572000
(Iteration 8001 / 24500) loss: 0.877207
(Epoch 17 / 50) train acc: 0.730000; val_acc: 0.585000
(Iteration 8501 / 24500) loss: 0.783398
(Epoch 18 / 50) train acc: 0.768000; val_acc: 0.578000
(Iteration 9001 / 24500) loss: 0.950474
(Epoch 19 / 50) train acc: 0.769000; val_acc: 0.596000
(Iteration 9501 / 24500) loss: 0.947860
(Epoch 20 / 50) train acc: 0.766000; val_acc: 0.585000
(Iteration 10001 / 24500) loss: 1.093720
(Epoch 21 / 50) train acc: 0.796000; val_acc: 0.574000
(Iteration 10501 / 24500) loss: 0.797550
(Epoch 22 / 50) train acc: 0.804000; val_acc: 0.583000
(Iteration 11001 / 24500) loss: 0.841176
(Epoch 23 / 50) train acc: 0.789000; val_acc: 0.578000
(Iteration 11501 / 24500) loss: 0.966470
(Epoch 24 / 50) train acc: 0.807000; val_acc: 0.577000
(Iteration 12001 / 24500) loss: 0.802871
(Epoch 25 / 50) train acc: 0.821000; val_acc: 0.587000
```

```
(Iteration 12501 / 24500) loss: 0.558691
(Epoch 26 / 50) train acc: 0.833000; val_acc: 0.606000
(Iteration 13001 / 24500) loss: 0.849243
(Epoch 27 / 50) train acc: 0.855000; val_acc: 0.580000
(Iteration 13501 / 24500) loss: 0.711638
(Epoch 28 / 50) train acc: 0.839000; val_acc: 0.608000
(Iteration 14001 / 24500) loss: 0.620865
(Epoch 29 / 50) train acc: 0.830000; val_acc: 0.595000
(Iteration 14501 / 24500) loss: 0.771924
(Epoch 30 / 50) train acc: 0.860000; val_acc: 0.577000
(Iteration 15001 / 24500) loss: 0.608603
(Epoch 31 / 50) train acc: 0.852000; val_acc: 0.600000
(Iteration 15501 / 24500) loss: 0.730044
(Epoch 32 / 50) train acc: 0.854000; val_acc: 0.594000
(Iteration 16001 / 24500) loss: 0.722283
(Epoch 33 / 50) train acc: 0.870000; val_acc: 0.610000
(Iteration 16501 / 24500) loss: 0.525413
(Epoch 34 / 50) train acc: 0.868000; val_acc: 0.601000
(Iteration 17001 / 24500) loss: 0.871143
(Epoch 35 / 50) train acc: 0.888000; val_acc: 0.592000
(Iteration 17501 / 24500) loss: 0.781858
(Epoch 36 / 50) train acc: 0.874000; val_acc: 0.593000
(Iteration 18001 / 24500) loss: 0.627746
(Epoch 37 / 50) train acc: 0.894000; val_acc: 0.606000
(Iteration 18501 / 24500) loss: 0.616153
(Epoch 38 / 50) train acc: 0.885000; val_acc: 0.604000
(Iteration 19001 / 24500) loss: 0.560498
(Epoch 39 / 50) train acc: 0.910000; val_acc: 0.600000
(Iteration 19501 / 24500) loss: 0.582070
(Epoch 40 / 50) train acc: 0.901000; val_acc: 0.593000
(Iteration 20001 / 24500) loss: 0.660565
(Epoch 41 / 50) train acc: 0.902000; val_acc: 0.600000
(Iteration 20501 / 24500) loss: 0.648878
(Epoch 42 / 50) train acc: 0.907000; val_acc: 0.620000
(Iteration 21001 / 24500) loss: 0.596133
(Epoch 43 / 50) train acc: 0.909000; val_acc: 0.590000
(Iteration 21501 / 24500) loss: 0.698394
(Epoch 44 / 50) train acc: 0.905000; val_acc: 0.598000
(Iteration 22001 / 24500) loss: 0.533412
(Epoch 45 / 50) train acc: 0.917000; val_acc: 0.599000
(Iteration 22501 / 24500) loss: 0.476177
(Epoch 46 / 50) train acc: 0.906000; val_acc: 0.604000
(Iteration 23001 / 24500) loss: 0.441500
(Epoch 47 / 50) train acc: 0.929000; val_acc: 0.593000
(Iteration 23501 / 24500) loss: 0.577921
(Epoch 48 / 50) train acc: 0.918000; val_acc: 0.609000
(Iteration 24001 / 24500) loss: 0.565005
(Epoch 49 / 50) train acc: 0.915000; val_acc: 0.605000
```

```
(Epoch 50 / 50) train acc: 0.904000; val_acc: 0.607000
CPU times: user 1h 17min 16s, sys: 35min 32s, total: 1h 52min 49s
Wall time: 57min 7s
```

1.7.7 Conclusion:

So after 1 hour of waiting, we finally did it! We have above 60% accuracy.

1.7.8 fc_net.py

[31]:
```python
import numpy as np
import pdb

from .layers import *
from .layer_utils import *

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

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

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

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

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

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

    Inputs:
```

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

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ============================================================ #
    self.params['W1'] = np.random.normal(scale=weight_scale, size = (input_dim,␣
↪hidden_dims), loc=0.0)
    self.params['W2'] = np.random.normal(scale=weight_scale, size =␣
↪(hidden_dims, num_classes), loc=0.0)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['b2'] = np.zeros(num_classes)

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

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

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

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

    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
```

```
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ============================================================ #
    W1, W2, b1, b2 = self.params['W1'], self.params['W2'], self.params['b1'],␣
↪self.params['b2']

    hidden, hidden_cache = affine_relu_forward(X, W1, b1)
    scores, scores_cache = affine_forward(hidden, W2, b2) #forward prop
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

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

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

    sum_term = np.sum(W1*W1) + np.sum(W2*W2)
    reg_term = 0.5 * self.reg * sum_term
    loss += reg_term

    dhidden, dW2, db2 = affine_backward(dscores, scores_cache)
    dX, dW1, db1 = affine_relu_backward(dhidden, hidden_cache)
```

```
    #update gradient

    grads['W1'] = self.reg * W1 + dW1
    grads['W2'] = self.reg * W2 + dW2
    grads['b1'] = db1
    grads['b2'] = db2
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

    return loss, grads


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

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

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

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

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

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden layer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify.
    - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch normalization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be performed using
```

```
    this datatype. float32 is faster but less accurate, so you should use
    float64 for numeric gradient checking.
  - seed: If not None, then pass this random seed to the dropout layers. This
    will make the dropout layers deteriminstic so we can gradient check the
    model.
  """
  self.use_batchnorm = use_batchnorm
  self.use_dropout = dropout > 0
  self.reg = reg
  self.num_layers = 1 + len(hidden_dims)
  self.dtype = dtype
  self.params = {}

  # ============================================================ #
  # YOUR CODE HERE:
  #   Initialize all parameters of the network in the self.params dictionary.
  #   The weights and biases of layer 1 are W1 and b1; and in general the
  #   weights and biases of layer i are Wi and bi. The
  #   biases are initialized to zero and the weights are initialized
  #   so that each parameter has mean 0 and standard deviation weight_scale.
  #
  #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
  #   parameters to zero.  The gamma and beta parameters for layer 1 should
  #   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
  #   should be gamma2 and beta2, etc. Only use batchnorm if self.
↪use_batchnorm
  #   is true and DO NOT do batch normalize the output scores.
  # ============================================================ #
  #start our loop here, going through all the layers
  for i in range(1, self.num_layers + 1):

      #add batchnorm
      if self.use_batchnorm and i != self.num_layers: #if i = self.
↪num_layers, index is out of range?
          gam = "gamma" + str(i)
          beta = "beta" + str(i)
          self.params[gam] = np.ones(hidden_dims[i-1])
          self.params[beta] = np.zeros(hidden_dims[i-1]) #0, 1 wouldn't work␣
↪lol

      #make our layers' names
      W = 'W' + str(i) #'W' + i wouldn't work
      b = 'b' + str(i)

      size, zeros_size = 0, 0

      #last layer
```

```
        if i == self.num_layers:
            size = (hidden_dims[i-2], num_classes)
            zeros_size = num_classes

        #first layer
        elif i == 1:
            size = (input_dim, hidden_dims[i-1])
            zeros_size = hidden_dims[i-1]

        #hidden layers
        else:
            size = (hidden_dims[i-2], hidden_dims[i-1])
            zeros_size = hidden_dims[i-1]

        #update params
        self.params[W] = np.random.normal(loc=0.0, scale=weight_scale,␣
↪size=size)
        self.params[b] = np.zeros(zeros_size)

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

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

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

    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)
```

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

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

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

  scores = None

  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the forward pass of the FC net and store the output
  #   scores as the variable "scores".
  #
  #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
  #   between the affine_forward and relu_forward layers.  You may
  #   also write an affine_batchnorm_relu() function in layer_utils.py.
  #
  #   DROPOUT: If dropout is non-zero, insert a dropout layer after
  #   every ReLU layer.
  # ============================================================ #
  hidden, hidden_cache, dropout_cache = [], [], []

  #same as in the init function
  for i in range(1, self.num_layers + 1):
      W = 'W' + str(i)
      b = 'b' + str(i)
      gam = 'gamma' + str(i)
      beta = "beta" + str(i)

      if i == self.num_layers:
          dup = affine_forward(hidden[i-2], self.params[W], self.params[b])
          scores, obj = dup

      else:

          #do hidden first
```

```
            if i == 1:
                hidden_param = X
            else:
                hidden_param = hidden[i-2]

            #try to prevent duplicate code
            if self.use_batchnorm :
                appendVar = affine_batchnorm_forward(hidden_param, self.
↪params[W], self.params[b], self.params[gam], self.params[beta], self.
↪bn_params[i-1])
            else:
                appendVar = affine_relu_forward(hidden_param, self.params[W],␣
↪self.params[b])

            hidden.append(appendVar[0])

            #then do hidden_cache
            obj = appendVar[1]

        hidden_cache.append(obj)

        #handle dropout after relu layer

        #self.use_dropout is in [0, 1]
        if i != self.num_layers and self.use_dropout > 0:

            hidden[i-1] = dropout_forward(hidden[i-1], self.dropout_param)[0]
            dropout_cache.append(dropout_forward(hidden[i-1], self.
↪dropout_param)[1])

            #below code is cleaner, but has more error
            #this is because we need to first modify hidden[i-1] (line 336)␣
↪before passing it into dropout_forward (line 337)
            #I didn't do that below, and hence my error is larger

#             dropoutAppendVar = dropout_forward(hidden[i-1], self.
↪dropout_param)

#             #update hidden layers

#             hidden[i-1] = dropoutAppendVar[0]

#             #update cache
#             dropout_cache.append(dropoutAppendVar[1])

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

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

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

    loss, grads = 0.0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backwards pass of the FC net and store the gradients
    #   in the grads dict, so that grads[k] is the gradient of self.params[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    #
    #   BATCHNORM: Incorporate the backward pass of the batchnorm.
    #
    #   DROPOUT: Incorporate the backward pass of dropout.
    # ============================================================ #
    loss, dscores = softmax_loss(scores, y)
    dhidden = [] #update this each iteration

    #go backwards
    for i in range(self.num_layers, 0, -1):
        W = 'W' + str(i)
        b = 'b' + str(i)
        gam = 'gamma' + str(i)
        beta = 'beta' + str(i)

        sum_term = self.params[W] * self.params[W]
        tot_term = 0.5 * self.reg * np.sum(sum_term)

        loss = loss + tot_term

        if i == self.num_layers:
            dh, grads[W], grads[b] = affine_backward(dscores, hidden_cache[self.
↪num_layers-1])
        else:

            #droppout happens before batchnorm / baseline NN backprop

            if self.use_dropout > 0:
                dh = dropout_backward(dh, dropout_cache[i-1])

            if self.use_batchnorm:
                dh, grads[W], grads[b], grads[gam], grads[beta] =␣
↪affine_batchnorm_backward(dh, hidden_cache[i-1])

            else:
```

```
                dh, grads[W], grads[b] = affine_relu_backward(dh,␣
↪hidden_cache[i-1])

        dhidden.append(dh)

        grads[W] += self.reg * self.params[W] #need to have tuple unpacking␣
↪above for this to work


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

    return loss, grads
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-31-d1db7b5b6ed6> in <module>()
      2 import pdb
      3
----> 4 from .layers import *
      5 from .layer_utils import *
      6

ModuleNotFoundError: No module named '__main__.layers'; '__main__' is not a␣
 ↪package

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.
To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

1.7.9 layers.py

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

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
```

```
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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

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

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

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ============================================================ #
  out = np.dot(x.reshape(x.shape[0], -1), w) + b
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #

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


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

  Inputs:
  - dout: Upstream derivative, of shape (N, M)
  - cache: Tuple of:
```

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

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

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

  db = np.sum(dout, axis=0)

  dw = np.dot(x.reshape(x.shape[0], -1).T, dout)

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

  return dx, dw, db

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

  Input:
  - x: Inputs, of any shape

  Returns a tuple of:
  - out: Output, of the same shape as x
  - cache: x
  """
  # ============================================================= #
```

```
  # YOUR CODE HERE:
  #   Implement the ReLU forward pass.
  # ================================================================ #
  relu = lambda x : x * (x > 0)
  out = relu(x)
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  cache = x
  return out, cache


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

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

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

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ================================================================ #
  # ReLU directs linearly to those > 0
  dx = dout * (x.reshape(x.shape[0], -1) >= 0)

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

  return dx

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

  During training the sample mean and (uncorrected) sample variance are
  computed from minibatch statistics and used to normalize the incoming data.
  During training we also keep an exponentially decaying running mean of the␣
 ↪mean
```

```
    and variance of each feature, and these averages are used to normalize data
    at test-time.

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

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

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

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

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

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype)) #key,␣
↪value to be returned if key is not found
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

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

      # ============================================================ #
      # YOUR CODE HERE:
      #   A few steps here:
      #     (1) Calculate the running mean and variance of the minibatch.
```

```
    #     (2) Normalize the activations with the sample mean and variance.␣
↪(changed running -> sample)
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass in
    #         the 'cache' variable.
    # ================================================================ #

    #what is the point of running_mean and running_var?? idk

    #from wk5_coding_dl video

    #step 1 -- calculate sample  mean + variance
    mu = np.mean(x, axis=0)
    sigma_squared = np.var(x, axis=0)

    #calculate running mean + variance
    running_mean = momentum * running_mean + (1.0 - momentum) * mu
    running_var = momentum * running_var + (1.0 - momentum) * sigma_squared

    #step 2 -- normalize with the *sample* mean and variance (Piazza @368)␣
↪[what does this mean?]
    xhat = (x - mu) / np.sqrt(sigma_squared + eps)

    #step 3 -- scale and shift

    #y -> out
    out = gamma*xhat + beta

    cache = (gamma, beta, running_var, running_mean, mu, sigma_squared, x, eps,␣
↪xhat)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
  elif mode == 'test':

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

    #Piazza #343 -> only 1 vector, so sample mean -> running mean
    xhat = (x - running_mean) / np.sqrt(running_var + eps)
    out = gamma*xhat + beta
```

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

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

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

  return out, cache

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

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

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

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
  # ============================================================ #
  gamma, beta, running_var, running_mean, mu, sigma_squared, x, eps, xhat =␣
↪cache #tuple unpacking

  N, D = dout.shape #rows, cols

  #from lecture slides: page 47 of lecture9_training, annotated

  dL_dxhat = dout * gamma

  temp1 = sigma_squared + eps
```

```
  #calculating dL_dsigma_squared
  first_part = - (x - mu) / (2 * temp1 ** (3/2))
  inside_sum = first_part * dL_dxhat

  #axis=0 means along the column, axis=1 means along the row; returns an array

  #when I got rid of axis=0, my error went from 10^-12 -> 1 lol
  dL_dsigma_squared = np.sum(inside_sum, axis=0)

  #calculated dL_dmu
  dL_dmu = - np.sum(dL_dxhat, axis=0) / np.sqrt(temp1)

  #calculate dL_dx = dx
  #in this case, m = N
  term1 = dL_dxhat / np.sqrt(temp1)
  term2 = 2 * (x - mu) * dL_dsigma_squared / N
  term3 = dL_dmu / N

  dx = term1 + term2 + term3

  #dbeta and dgamma can be computed from page 44 of lecture9_training slides
  dbeta = np.sum(dout, axis=0)
  dgamma = np.sum(dout*xhat, axis=0) #is axis=0 even needed?

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

  return dx, dgamma, dbeta

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

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

  Outputs:
  - out: Array of the same shape as x.
```

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

  mask = None
  out = None

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

    mask = (np.random.rand(*x.shape) < p) / p #should we normalize here?

    out = mask * x #commutative since * in numpy is the Hadamard product (entry␣
↪wise)

    #print("mask shape:", mask.shape)
    #print("out shape:", out.shape)

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

  elif mode == 'test':

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

    #don't modify mask, since mask = None
    out = x

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

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

```
  return out, cache

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

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

  dx = None
  if mode == 'train':
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during training time.
    # ============================================================ #
    dx = dout * mask #piazza #367
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
  elif mode == 'test':
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the inverted dropout backward pass during test time.
    # ============================================================ #
    dx = dout
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
  return dx

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

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

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

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


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

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

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

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