ECE C147/247 HW4 Q1: 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.

`utils` 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`.

In [2]:
```
## Import and setups

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

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

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

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

In [3]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

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`

Test all functions you copy and pasted

In [4]:
```
from nndl.layer_tests import *

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

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


If affine_backward is working, error should be less than 1e-9::
dx error: 4.616315416l927085e-10
dw error: 6.707494642048358e-11
db error: 6.942320637660904e-12


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: 3.2756279540991835e-12


If affine_relu_forward and affine_relu_backward are working, error should be less than 1e-9::
dx error: 5.524297922807348e-11
dw error: 3.631734171907549e-10
db error: 3.2756041823344864e-12


Running check with reg = 0
Initial loss: 2.303392659269333
W1 relative error: 2.865557280343655e-05
W2 relative error: 4.988612955448939e-07
W3 relative error: 8.249390118912997e-08
b1 relative error: 3.68580394215281e-07
b2 relative error: 3.4440948223621555e-09
b3 relative error: 1.4614235824389521e-10
Running check with reg = 3.14
Initial loss: 5.744233115944262
W1 relative error: 4.160769835235782e-09
W2 relative error: 2.7898679841333852e-08
W3 relative error: 1.0
b1 relative error: 2.234842945592196e-06
b2 relative error: 1.1539627622763397e-08
b3 relative error: 2.6543776080429063e-10
```

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.

SGD + momentum

In the following section, implement SGD with momentum. Read the `nndl/optim.py` API, 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.

In [5]:
```
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
```

SGD + Nesterov momentum

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

In [6]:
```
from nndl.optim import sgd_nesterov_momentum

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

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

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

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

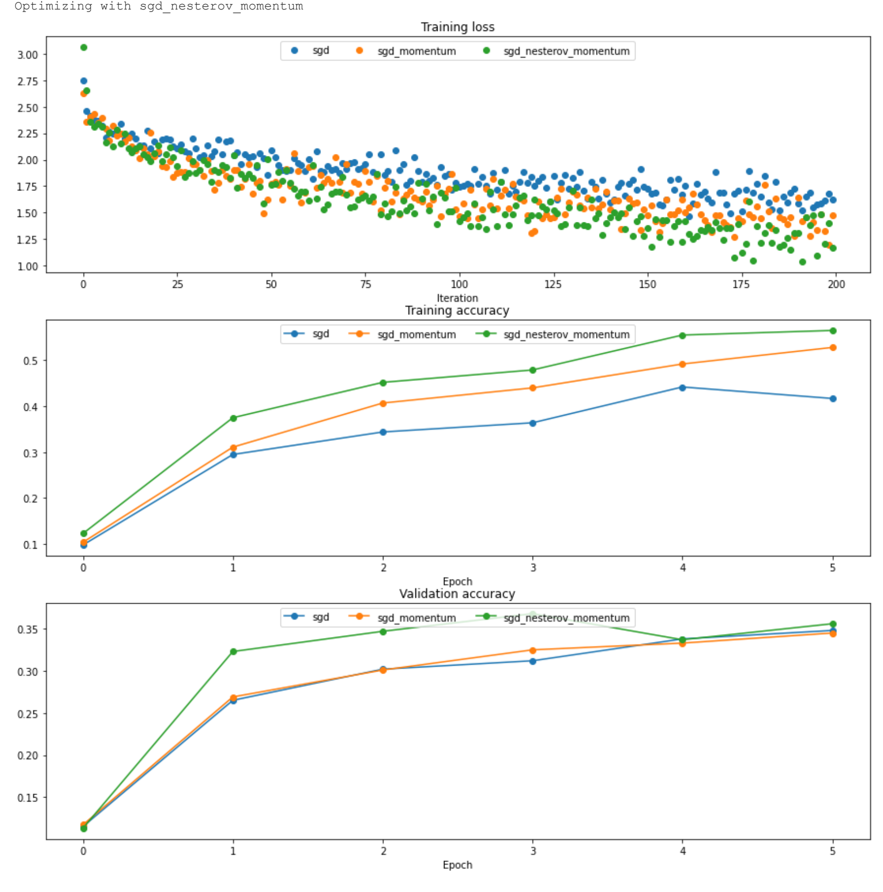
```
next_w error: 1.0875186845081027e-08
velocity error: 4.269287743278663e-09
```

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.

In [7]:
```
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
```



RMSProp

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

In [8]:
```
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: 9.502645229894295e-08
cache error: 2.6477955807156126e-09
```

Adaptive moments

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

In [9]:
```
# 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'])))
```

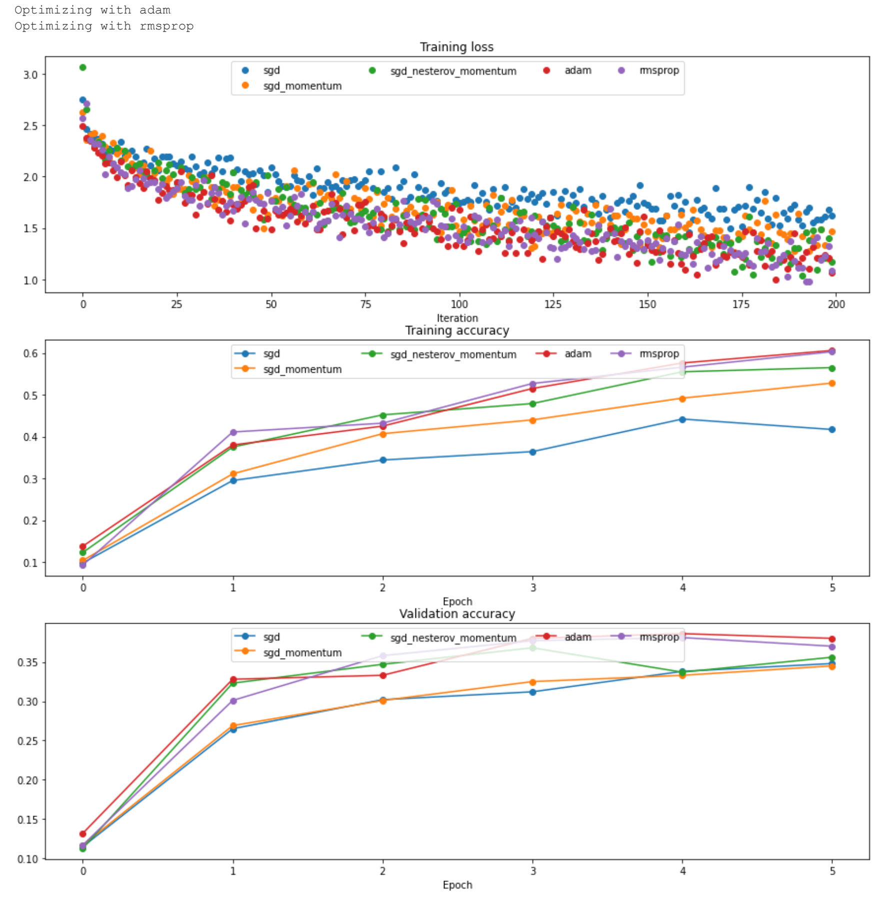
```
next_w error: 1.1395691798535431e-07
a error: 4.208314038113071e-09
v error: 4.214963193114416e-09
```

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.

In [10]:
```
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
```



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.

In [11]:
```
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: 2.324108
(Epoch 0 / 10) train acc: 0.259000; val_acc: 0.236000
(Iteration 51 / 4900) loss: 1.710812
(Iteration 101 / 4900) loss: 1.569633
(Iteration 151 / 4900) loss: 1.704779
(Iteration 201 / 4900) loss: 1.553350
(Iteration 251 / 4900) loss: 1.622926
(Iteration 301 / 4900) loss: 1.565025
(Iteration 351 / 4900) loss: 1.365191
(Iteration 401 / 4900) loss: 1.479973
(Iteration 451 / 4900) loss: 1.471955
(Epoch 1 / 10) train acc: 0.469000; val_acc: 0.480000
(Iteration 501 / 4900) loss: 1.354020
(Iteration 551 / 4900) loss: 1.429286
(Iteration 601 / 4900) loss: 1.348315
(Iteration 651 / 4900) loss: 1.361091
(Iteration 701 / 4900) loss: 1.366582
(Iteration 751 / 4900) loss: 1.391804
(Iteration 801 / 4900) loss: 1.291474
(Iteration 851 / 4900) loss: 1.355819
(Iteration 901 / 4900) loss: 1.189257
(Iteration 951 / 4900) loss: 1.358157
(Epoch 2 / 10) train acc: 0.587000; val_acc: 0.509000
(Iteration 1001 / 4900) loss: 1.190631
(Iteration 1051 / 4900) loss: 1.337119
(Iteration 1101 / 4900) loss: 1.135299
(Iteration 1151 / 4900) loss: 1.194532
(Iteration 1201 / 4900) loss: 1.087486
(Iteration 1251 / 4900) loss: 1.280915
(Iteration 1301 / 4900) loss: 1.267049
(Iteration 1351 / 4900) loss: 1.082300
(Iteration 1401 / 4900) loss: 0.998963
(Iteration 1451 / 4900) loss: 0.952096
(Epoch 3 / 10) train acc: 0.603000; val_acc: 0.516000
(Iteration 1501 / 4900) loss: 1.180900
(Iteration 1551 / 4900) loss: 1.246343
(Iteration 1601 / 4900) loss: 1.180023
(Iteration 1651 / 4900) loss: 1.222134
(Iteration 1701 / 4900) loss: 0.959548
(Iteration 1751 / 4900) loss: 1.044596
(Iteration 1801 / 4900) loss: 0.926830
(Iteration 1851 / 4900) loss: 1.047942
(Iteration 1901 / 4900) loss: 0.988956
(Iteration 1951 / 4900) loss: 1.044814
(Epoch 4 / 10) train acc: 0.616000; val_acc: 0.533000
(Iteration 2001 / 4900) loss: 1.002781
(Iteration 2051 / 4900) loss: 0.906200
(Iteration 2101 / 4900) loss: 1.015182
(Iteration 2151 / 4900) loss: 0.796182
(Iteration 2201 / 4900) loss: 0.831100
(Iteration 2251 / 4900) loss: 0.943659
(Iteration 2301 / 4900) loss: 0.893647
(Iteration 2351 / 4900) loss: 1.119650
(Iteration 2401 / 4900) loss: 0.897629
(Epoch 5 / 10) train acc: 0.675000; val_acc: 0.543000
(Iteration 2451 / 4900) loss: 1.015501
(Iteration 2501 / 4900) loss: 0.888721
(Iteration 2551 / 4900) loss: 0.900259
(Iteration 2601 / 4900) loss: 1.006299
(Iteration 2651 / 4900) loss: 0.866825
(Iteration 2701 / 4900) loss: 0.864173
(Iteration 2751 / 4900) loss: 0.882573
(Iteration 2801 / 4900) loss: 0.896280
(Iteration 2851 / 4900) loss: 0.950030
(Iteration 2901 / 4900) loss: 0.898759
(Epoch 6 / 10) train acc: 0.714000; val_acc: 0.547000
(Iteration 2951 / 4900) loss: 0.630894
(Iteration 3001 / 4900) loss: 0.776758
(Iteration 3051 / 4900) loss: 0.717995
(Iteration 3101 / 4900) loss: 0.637762
(Iteration 3151 / 4900) loss: 0.797481
(Iteration 3201 / 4900) loss: 0.783366
(Iteration 3251 / 4900) loss: 0.792586
(Iteration 3301 / 4900) loss: 0.918864
(Iteration 3351 / 4900) loss: 0.820589
(Iteration 3401 / 4900) loss: 0.743079
(Epoch 7 / 10) train acc: 0.739000; val_acc: 0.581000
(Iteration 3451 / 4900) loss: 0.877669
(Iteration 3501 / 4900) loss: 0.937903
(Iteration 3551 / 4900) loss: 0.782537
(Iteration 3601 / 4900) loss: 0.855775
(Iteration 3651 / 4900) loss: 0.653582
(Iteration 3701 / 4900) loss: 0.745576
(Iteration 3751 / 4900) loss: 0.787852
(Iteration 3801 / 4900) loss: 0.702431
(Iteration 3851 / 4900) loss: 0.762737
(Iteration 3901 / 4900) loss: 0.668990
(Epoch 8 / 10) train acc: 0.740000; val_acc: 0.539000
(Iteration 3951 / 4900) loss: 0.585486
(Iteration 4001 / 4900) loss: 0.665097
(Iteration 4051 / 4900) loss: 0.682409
(Iteration 4101 / 4900) loss: 0.759852
(Iteration 4151 / 4900) loss: 0.621568
(Iteration 4201 / 4900) loss: 0.445623
(Iteration 4251 / 4900) loss: 0.703892
(Iteration 4301 / 4900) loss: 0.804312
(Iteration 4351 / 4900) loss: 0.650961
(Iteration 4401 / 4900) loss: 0.528363
(Epoch 9 / 10) train acc: 0.801000; val_acc: 0.573000
(Iteration 4451 / 4900) loss: 0.626053
(Iteration 4501 / 4900) loss: 0.512020
(Iteration 4551 / 4900) loss: 0.480116
(Iteration 4601 / 4900) loss: 0.631010
(Iteration 4651 / 4900) loss: 0.762133
(Iteration 4701 / 4900) loss: 0.537167
(Iteration 4751 / 4900) loss: 0.638203
(Iteration 4801 / 4900) loss: 0.516493
(Iteration 4851 / 4900) loss: 0.517716
(Epoch 10 / 10) train acc: 0.830000; val_acc: 0.558000
```

In [12]:
```
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: 0.584
Test set accuracy: 0.553
```

In [ ]:

ECE C147/247 HW4 Q2: 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.

`utils` 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`.

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

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

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

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

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

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

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

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

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.

In [3]:
```
# 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:  [49.25890874 15.68042077 10.77425698]
  stds:  [32.74515657 29.35640593 37.42221779]
After batch normalization (gamma=1, beta=0)
  mean:  [-2.54796184e-16 -4.71844785e-17 -3.72618603e-17]
  std:  [1.         0.99999999 1.        ]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [1.         1.99999999 2.99999999]
```

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

In [4]:
```
# 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:  [ 0.11609137 -0.09502236  0.07165287]
  stds:  [1.05963918 0.92295591 0.99428131]
```

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.

In [5]:
```
# 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:  5.586527822853507e-10
dgamma error:  2.6632508082038723e-11
dbeta error:  4.977243498503734e-11
```

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.

In [6]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for 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:  2.3214776413153135
W1 relative error: 1.701456185897653e-05
W2 relative error: 5.150159402515397e-06
W3 relative error: 5.277463191023611e-10
b1 relative error: 0.0022204516003654358
b2 relative error: 5.551115123125783e-09
b3 relative error: 1.1705603130984187e-10
beta1 relative error: 4.104526065485512e-09
beta2 relative error: 1.0109375534584169e-08
gamma1 relative error: 3.2168407209000194e-09
gamma2 relative error: 5.788857349186422e-09


Running check with reg =  3.14
Initial loss:  5.961655036876346
W1 relative error: 3.338186289789542e-05
W2 relative error: 1.2190849287679875e-06
W3 relative error: 1.0
b1 relative error: 5.551115123125783e-09
b2 relative error: 4.440892098500626e-08
b3 relative error: 2.847632039326928e-10
beta1 relative error: 7.358215383341911e-09
beta2 relative error: 5.591805549094429e-09
gamma1 relative error: 7.616191542918951e-09
gamma2 relative error: 1.1586508381957641e-08
```

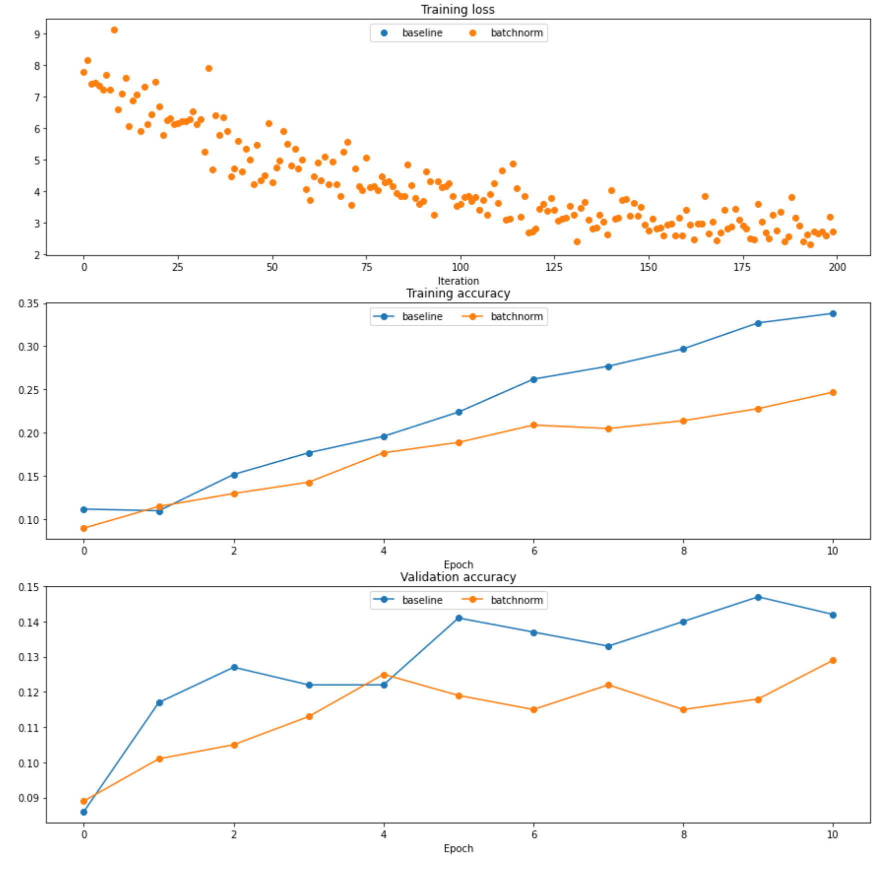
Training a deep fully connected network with batch normalization.

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

In [7]:
```
# 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: 2.294970
(Epoch 0 / 10) train acc: 0.157000; val_acc: 0.134000
(Epoch 1 / 10) train acc: 0.357000; val_acc: 0.272000
(Epoch 2 / 10) train acc: 0.407000; val_acc: 0.309000
(Epoch 3 / 10) train acc: 0.477000; val_acc: 0.338000
(Epoch 4 / 10) train acc: 0.539000; val_acc: 0.323000
(Epoch 5 / 10) train acc: 0.628000; val_acc: 0.322000
(Epoch 6 / 10) train acc: 0.669000; val_acc: 0.309000
(Epoch 7 / 10) train acc: 0.680000; val_acc: 0.309000
(Epoch 8 / 10) train acc: 0.668000; val_acc: 0.308000
(Epoch 9 / 10) train acc: 0.726000; val_acc: 0.334000
(Epoch 10 / 10) train acc: 0.795000; val_acc: 0.327000
(Iteration 1 / 200) loss: 2.302627
(Epoch 0 / 10) train acc: 0.163000; val_acc: 0.149000
(Epoch 1 / 10) train acc: 0.216000; val_acc: 0.201000
(Epoch 2 / 10) train acc: 0.304000; val_acc: 0.249000
(Epoch 3 / 10) train acc: 0.320000; val_acc: 0.279000
(Epoch 4 / 10) train acc: 0.350000; val_acc: 0.246000
(Epoch 5 / 10) train acc: 0.462000; val_acc: 0.334000
(Epoch 6 / 10) train acc: 0.469000; val_acc: 0.298000
(Epoch 7 / 10) train acc: 0.546000; val_acc: 0.320000
(Epoch 8 / 10) train acc: 0.569000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.605000; val_acc: 0.336000
(Epoch 10 / 10) train acc: 0.660000; val_acc: 0.354000
```

In [10]:
```
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()
```



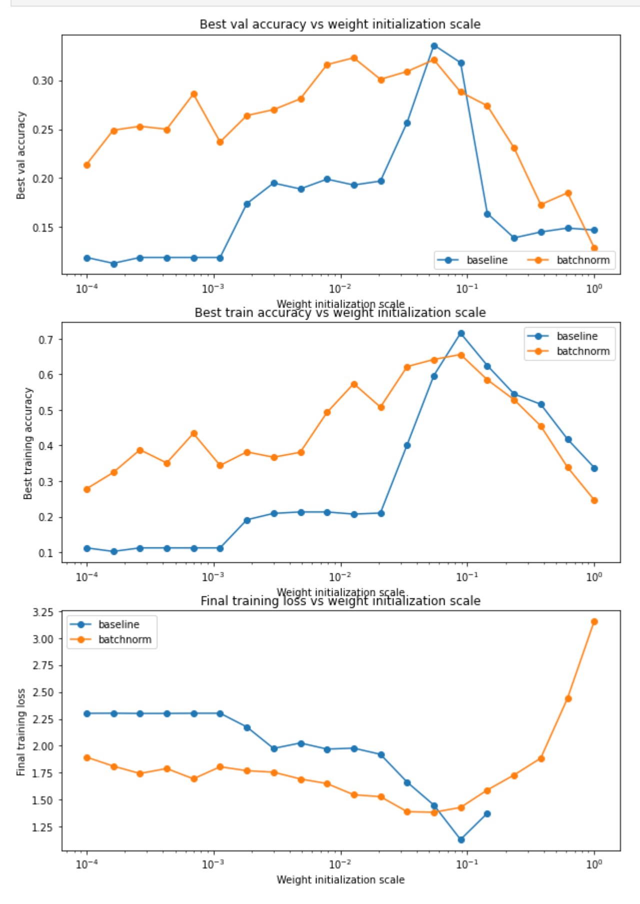
Batchnorm and initialization

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.

In [9]:
```
# 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
/Users/jack_tseng/Documents/UCLA/Courses/winter_2021/c247/hw/hw4/assignment/hw4-code/nndl/layers.py:426: 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
```

In [11]:
```
# 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()
```



Question:

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

Answer:

Batchnorm would help stabilize the input data, also keep each backpropagations less effect by previous one. This would decrease the number of the required epoch. As we can see in the graph, while the epoch is small, batchnorm technique has better performance (~30% better). However, the benifits from batchnorm might vanish while the epoch is large.

ECE C147/247 HW4 Q3: 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.

`utils` 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`.

In [1]:

```python
## Import and setups

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

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

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

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

In [2]:

```python
# 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,)
```

Dropout forward pass

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

In [4]:

```python
x = np.random.randn(500, 500) + 10

for p in [0.3, 0.6, 0.75]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
```

```
Running tests with p =  0.3
Mean of input:  10.000557087358972
Mean of train-time output:  10.022214562820208
Mean of test-time output:  9.985855217232421
Fraction of train-time output set to zero:  0.699352
Fraction of test-time output set to zero:  0.301104
Running tests with p =  0.6
Mean of input:  10.000557087358972
Mean of train-time output:  9.985770206712672
Mean of test-time output:  10.002784182816438
Fraction of train-time output set to zero:  0.400944
Fraction of test-time output set to zero:  0.600004
Running tests with p =  0.75
Mean of input:  10.000557087358972
Mean of train-time output:  9.973948993078066
Mean of test-time output:  9.990809148589877
Fraction of train-time output set to zero:  0.252104
Fraction of test-time output set to zero:  0.750196
```

Dropout backward pass

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

In [5]:

```python
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param)[0], x, dout)

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

```
dx relative error:  5.445613019215856e-11
```

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).

In [6]:

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

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

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

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

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


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


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

Dropout as a regularizer

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

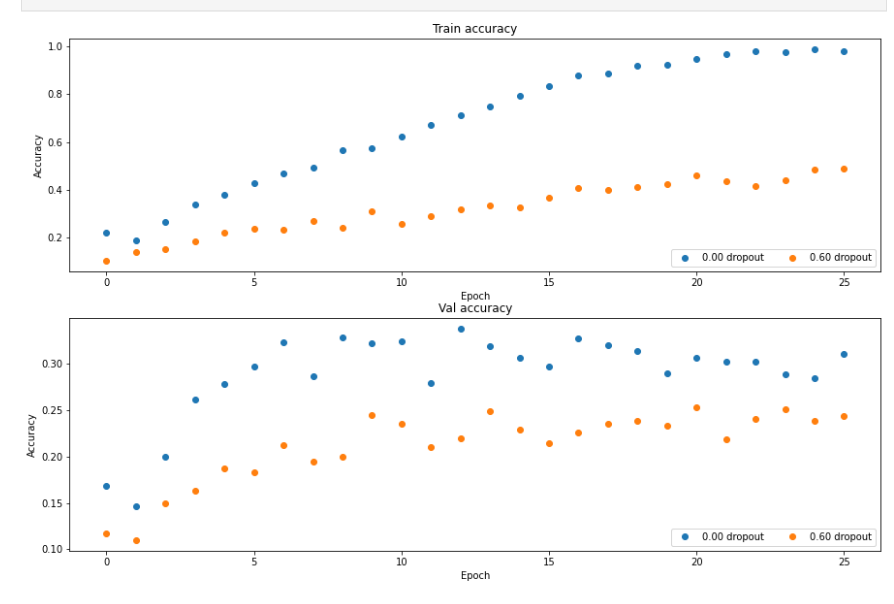
In [7]:

```python
# Train two identical nets, one with dropout and one without

num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [0, 0.6]
for dropout in dropout_choices:
  model = FullyConnectedNet([100, 100, 100], dropout=dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

```
(Iteration 1 / 125) loss: 2.300804
(Epoch 0 / 25) train acc: 0.220000; val_acc: 0.168000
(Epoch 1 / 25) train acc: 0.188000; val_acc: 0.147000
(Epoch 2 / 25) train acc: 0.266000; val_acc: 0.200000
(Epoch 3 / 25) train acc: 0.338000; val_acc: 0.262000
(Epoch 4 / 25) train acc: 0.378000; val_acc: 0.278000
(Epoch 5 / 25) train acc: 0.428000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.468000; val_acc: 0.323000
(Epoch 7 / 25) train acc: 0.494000; val_acc: 0.287000
(Epoch 8 / 25) train acc: 0.566000; val_acc: 0.328000
(Epoch 9 / 25) train acc: 0.572000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.622000; val_acc: 0.324000
(Epoch 11 / 25) train acc: 0.670000; val_acc: 0.279000
(Epoch 12 / 25) train acc: 0.710000; val_acc: 0.338000
(Epoch 13 / 25) train acc: 0.746000; val_acc: 0.319000
(Epoch 14 / 25) train acc: 0.792000; val_acc: 0.307000
(Epoch 15 / 25) train acc: 0.834000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.876000; val_acc: 0.327000
(Epoch 17 / 25) train acc: 0.886000; val_acc: 0.320000
(Epoch 18 / 25) train acc: 0.918000; val_acc: 0.314000
(Epoch 19 / 25) train acc: 0.922000; val_acc: 0.290000
(Epoch 20 / 25) train acc: 0.944000; val_acc: 0.306000
(Iteration 101 / 125) loss: 0.156105
(Epoch 21 / 25) train acc: 0.968000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.302000
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.986000; val_acc: 0.285000
(Epoch 25 / 25) train acc: 0.978000; val_acc: 0.311000
(Iteration 1 / 125) loss: 2.301328
(Epoch 0 / 25) train acc: 0.104000; val_acc: 0.117000
(Epoch 1 / 25) train acc: 0.142000; val_acc: 0.110000
(Epoch 2 / 25) train acc: 0.154000; val_acc: 0.150000
(Epoch 3 / 25) train acc: 0.184000; val_acc: 0.163000
(Epoch 4 / 25) train acc: 0.222000; val_acc: 0.187000
(Epoch 5 / 25) train acc: 0.238000; val_acc: 0.183000
(Epoch 6 / 25) train acc: 0.232000; val_acc: 0.212000
(Epoch 7 / 25) train acc: 0.270000; val_acc: 0.195000
(Epoch 8 / 25) train acc: 0.240000; val_acc: 0.200000
(Epoch 9 / 25) train acc: 0.312000; val_acc: 0.245000
(Epoch 10 / 25) train acc: 0.258000; val_acc: 0.235000
(Epoch 11 / 25) train acc: 0.290000; val_acc: 0.210000
(Epoch 12 / 25) train acc: 0.318000; val_acc: 0.220000
(Epoch 13 / 25) train acc: 0.334000; val_acc: 0.249000
(Epoch 14 / 25) train acc: 0.328000; val_acc: 0.229000
(Epoch 15 / 25) train acc: 0.368000; val_acc: 0.214000
(Epoch 16 / 25) train acc: 0.408000; val_acc: 0.226000
(Epoch 17 / 25) train acc: 0.400000; val_acc: 0.235000
(Epoch 18 / 25) train acc: 0.412000; val_acc: 0.239000
(Epoch 19 / 25) train acc: 0.426000; val_acc: 0.233000
(Epoch 20 / 25) train acc: 0.460000; val_acc: 0.253000
(Iteration 101 / 125) loss: 1.158332
(Epoch 21 / 25) train acc: 0.436000; val_acc: 0.219000
(Epoch 22 / 25) train acc: 0.416000; val_acc: 0.241000
(Epoch 23 / 25) train acc: 0.442000; val_acc: 0.251000
(Epoch 24 / 25) train acc: 0.486000; val_acc: 0.239000
(Epoch 25 / 25) train acc: 0.488000; val_acc: 0.244000
```

In [8]:

```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()
```



Question

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

Answer:

Yes, it is. As the dropout being implemented, the training accuracy start decreasing, but the testing accuracy start increasing.

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.

In [13]:

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

# Choosing adam optimizer as professor recommend during the class (for small size model)
optim = 'adam'
dropout = 0.7
num_layers = 4
hidden_dims = [500] * num_layers
weight_scale = 0.01
learning_rate = 1e-4
lr_decay = 0.92

model = FullyConnectedNet(hidden_dims,
                          weight_scale=weight_scale,
                          dropout=dropout,
                          use_batchnorm=True)
solver = Solver(model,
                data,
                num_epochs=50,
                batch_size=100,
                update_rule=optim,
                optim_config={'learning_rate': learning_rate},
                lr_decay=lr_decay,
                verbose=True,
                print_every=100
               )

solver.train()


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

```
(Iteration 1 / 24500) loss: 2.317118
(Epoch 0 / 50) train acc: 0.113000; val_acc: 0.083000
(Iteration 101 / 24500) loss: 2.019870
(Iteration 201 / 24500) loss: 1.578769
(Iteration 301 / 24500) loss: 1.554016
(Iteration 401 / 24500) loss: 1.791607
(Epoch 1 / 50) train acc: 0.280000; val_acc: 0.270000
(Iteration 501 / 24500) loss: 1.579272
(Iteration 601 / 24500) loss: 1.622223
(Iteration 701 / 24500) loss: 1.494899
(Iteration 801 / 24500) loss: 1.369107
(Iteration 901 / 24500) loss: 1.461568
(Epoch 2 / 50) train acc: 0.322000; val_acc: 0.313000
(Iteration 1001 / 24500) loss: 1.577887
(Iteration 1101 / 24500) loss: 1.660637
(Iteration 1201 / 24500) loss: 1.474895
(Iteration 1301 / 24500) loss: 1.600894
(Iteration 1401 / 24500) loss: 1.312025
(Epoch 3 / 50) train acc: 0.338000; val_acc: 0.310000
(Iteration 1501 / 24500) loss: 1.244318
(Iteration 1601 / 24500) loss: 1.439889
(Iteration 1701 / 24500) loss: 1.506529
(Iteration 1801 / 24500) loss: 1.363044
(Iteration 1901 / 24500) loss: 1.247146
(Epoch 4 / 50) train acc: 0.364000; val_acc: 0.333000
(Iteration 2001 / 24500) loss: 1.258044
(Iteration 2101 / 24500) loss: 1.206013
(Iteration 2201 / 24500) loss: 1.523043
(Iteration 2301 / 24500) loss: 1.184978
(Iteration 2401 / 24500) loss: 1.334218
(Epoch 5 / 50) train acc: 0.355000; val_acc: 0.333000
(Iteration 2501 / 24500) loss: 1.375534
(Iteration 2601 / 24500) loss: 1.302815
(Iteration 2701 / 24500) loss: 1.269757
(Iteration 2801 / 24500) loss: 1.196946
(Iteration 2901 / 24500) loss: 1.327286
(Epoch 6 / 50) train acc: 0.352000; val_acc: 0.332000
(Iteration 3001 / 24500) loss: 1.163420
(Iteration 3101 / 24500) loss: 1.279980
(Iteration 3201 / 24500) loss: 1.278635
(Iteration 3301 / 24500) loss: 1.274505
(Iteration 3401 / 24500) loss: 1.164110
(Epoch 7 / 50) train acc: 0.361000; val_acc: 0.356000
(Iteration 3501 / 24500) loss: 1.209167
(Iteration 3601 / 24500) loss: 1.320131
(Iteration 3701 / 24500) loss: 1.188140
(Iteration 3801 / 24500) loss: 1.246226
(Iteration 3901 / 24500) loss: 1.351732
(Epoch 8 / 50) train acc: 0.368000; val_acc: 0.333000
(Iteration 4001 / 24500) loss: 1.224672
(Iteration 4101 / 24500) loss: 1.160422
(Iteration 4201 / 24500) loss: 1.154487
(Iteration 4301 / 24500) loss: 1.143211
(Iteration 4401 / 24500) loss: 1.190719
(Epoch 9 / 50) train acc: 0.367000; val_acc: 0.354000
(Iteration 4501 / 24500) loss: 1.214232
(Iteration 4601 / 24500) loss: 1.172209
(Iteration 4701 / 24500) loss: 1.103842
(Iteration 4801 / 24500) loss: 1.344881
(Epoch 10 / 50) train acc: 0.377000; val_acc: 0.341000
(Iteration 4901 / 24500) loss: 1.173015
(Iteration 5001 / 24500) loss: 1.103642
(Iteration 5101 / 24500) loss: 1.338772
(Iteration 5201 / 24500) loss: 0.999570
(Iteration 5301 / 24500) loss: 1.129549
(Epoch 11 / 50) train acc: 0.375000; val_acc: 0.343000
(Iteration 5401 / 24500) loss: 1.202527
(Iteration 5501 / 24500) loss: 1.088772
(Iteration 5601 / 24500) loss: 1.080738
(Iteration 5701 / 24500) loss: 1.278490
(Iteration 5801 / 24500) loss: 1.023285
(Epoch 12 / 50) train acc: 0.383000; val_acc: 0.363000
(Iteration 5901 / 24500) loss: 1.122243
(Iteration 6001 / 24500) loss: 1.131678
(Iteration 6101 / 24500) loss: 1.285635
(Iteration 6201 / 24500) loss: 0.994744
(Iteration 6301 / 24500) loss: 0.995584
(Epoch 13 / 50) train acc: 0.375000; val_acc: 0.371000
(Iteration 6401 / 24500) loss: 1.148755
(Iteration 6501 / 24500) loss: 1.000689
(Iteration 6601 / 24500) loss: 0.950830
(Iteration 6701 / 24500) loss: 0.939583
(Iteration 6801 / 24500) loss: 1.133951
(Epoch 14 / 50) train acc: 0.367000; val_acc: 0.354000
(Iteration 6901 / 24500) loss: 1.175240
(Iteration 7001 / 24500) loss: 1.112139
(Iteration 7101 / 24500) loss: 0.966774
(Iteration 7201 / 24500) loss: 1.031363
(Iteration 7301 / 24500) loss: 1.076373
(Epoch 15 / 50) train acc: 0.377000; val_acc: 0.335000
(Iteration 7401 / 24500) loss: 0.987290
(Iteration 7501 / 24500) loss: 1.174035
(Iteration 7601 / 24500) loss: 1.070008
(Iteration 7701 / 24500) loss: 1.030783
(Iteration 7801 / 24500) loss: 1.062699
(Epoch 16 / 50) train acc: 0.399000; val_acc: 0.374000
(Iteration 7901 / 24500) loss: 1.200432
(Iteration 8001 / 24500) loss: 0.939683
(Iteration 8101 / 24500) loss: 0.966704
(Iteration 8201 / 24500) loss: 0.868019
(Iteration 8301 / 24500) loss: 0.980183
(Epoch 17 / 50) train acc: 0.392000; val_acc: 0.327000
(Iteration 8401 / 24500) loss: 1.115581
(Iteration 8501 / 24500) loss: 1.007735
(Iteration 8601 / 24500) loss: 1.132872
(Iteration 8701 / 24500) loss: 1.078555
(Iteration 8801 / 24500) loss: 0.942440
(Epoch 18 / 50) train acc: 0.386000; val_acc: 0.350000
(Iteration 8901 / 24500) loss: 1.102465
(Iteration 9001 / 24500) loss: 1.067918
(Iteration 9101 / 24500) loss: 1.135233
(Iteration 9201 / 24500) loss: 0.962111
(Iteration 9301 / 24500) loss: 1.108376
(Epoch 19 / 50) train acc: 0.369000; val_acc: 0.372000
(Iteration 9401 / 24500) loss: 1.250284
(Iteration 9501 / 24500) loss: 1.063336
(Iteration 9601 / 24500) loss: 0.763866
(Iteration 9701 / 24500) loss: 1.105096
(Epoch 20 / 50) train acc: 0.407000; val_acc: 0.348000
(Iteration 9801 / 24500) loss: 0.877451
(Iteration 9901 / 24500) loss: 1.043127
(Iteration 10001 / 24500) loss: 0.815638
(Iteration 10101 / 24500) loss: 1.133042
(Iteration 10201 / 24500) loss: 1.244170
(Epoch 21 / 50) train acc: 0.417000; val_acc: 0.352000
(Iteration 10301 / 24500) loss: 0.927840
(Iteration 10401 / 24500) loss: 1.128871
(Iteration 10501 / 24500) loss: 0.999999
(Iteration 10601 / 24500) loss: 1.040392
(Iteration 10701 / 24500) loss: 0.935715
(Epoch 22 / 50) train acc: 0.387000; val_acc: 0.340000
(Iteration 10801 / 24500) loss: 1.031445
(Iteration 10901 / 24500) loss: 1.177752
(Iteration 11001 / 24500) loss: 0.876199
(Iteration 11101 / 24500) loss: 0.832888
(Iteration 11201 / 24500) loss: 0.977458
(Epoch 23 / 50) train acc: 0.413000; val_acc: 0.331000
(Iteration 11301 / 24500) loss: 0.815665
(Iteration 11401 / 24500) loss: 0.884473
(Iteration 11501 / 24500) loss: 0.999002
(Iteration 11601 / 24500) loss: 1.047803
(Iteration 11701 / 24500) loss: 1.149156
(Epoch 24 / 50) train acc: 0.368000; val_acc: 0.353000
(Iteration 11801 / 24500) loss: 0.965166
(Iteration 11901 / 24500) loss: 0.926483
(Iteration 12001 / 24500) loss: 1.038117
(Iteration 12101 / 24500) loss: 0.965281
(Iteration 12201 / 24500) loss: 0.812142
(Epoch 25 / 50) train acc: 0.384000; val_acc: 0.343000
(Iteration 12301 / 24500) loss: 1.134963
(Iteration 12401 / 24500) loss: 0.816007
(Iteration 12501 / 24500) loss: 0.778373
(Iteration 12601 / 24500) loss: 1.169505
(Iteration 12701 / 24500) loss: 1.038036
(Epoch 26 / 50) train acc: 0.381000; val_acc: 0.330000
(Iteration 12801 / 24500) loss: 1.106328
(Iteration 12901 / 24500) loss: 0.920046
(Iteration 13001 / 24500) loss: 0.947364
(Iteration 13101 / 24500) loss: 1.027788
(Iteration 13201 / 24500) loss: 0.873738
(Epoch 27 / 50) train acc: 0.419000; val_acc: 0.341000
(Iteration 13301 / 24500) loss: 0.845349
(Iteration 13401 / 24500) loss: 0.972021
(Iteration 13501 / 24500) loss: 0.861391
(Iteration 13601 / 24500) loss: 1.022596
(Iteration 13701 / 24500) loss: 1.157520
(Epoch 28 / 50) train acc: 0.380000; val_acc: 0.347000
(Iteration 13801 / 24500) loss: 0.808689
(Iteration 13901 / 24500) loss: 0.847242
(Iteration 14001 / 24500) loss: 0.837238
(Iteration 14101 / 24500) loss: 1.057651
(Iteration 14201 / 24500) loss: 0.856506
(Epoch 29 / 50) train acc: 0.402000; val_acc: 0.336000
(Iteration 14301 / 24500) loss: 0.934347
(Iteration 14401 / 24500) loss: 0.830593
(Iteration 14501 / 24500) loss: 0.933022
(Iteration 14601 / 24500) loss: 1.050151
(Epoch 30 / 50) train acc: 0.386000; val_acc: 0.322000
(Iteration 14701 / 24500) loss: 1.024442
(Iteration 14801 / 24500) loss: 1.138108
(Iteration 14901 / 24500) loss: 1.050816
(Iteration 15001 / 24500) loss: 1.037709
(Iteration 15101 / 24500) loss: 0.928268
(Epoch 31 / 50) train acc: 0.398000; val_acc: 0.331000
(Iteration 15201 / 24500) loss: 1.019207
(Iteration 15301 / 24500) loss: 0.938588
(Iteration 15401 / 24500) loss: 1.022457
(Iteration 15501 / 24500) loss: 0.896657
(Iteration 15601 / 24500) loss: 1.095885
(Epoch 32 / 50) train acc: 0.389000; val_acc: 0.344000
(Iteration 15701 / 24500) loss: 1.026435
(Iteration 15801 / 24500) loss: 0.961773
(Iteration 15901 / 24500) loss: 0.949461
(Iteration 16001 / 24500) loss: 0.857172
(Iteration 16101 / 24500) loss: 0.925160
(Epoch 33 / 50) train acc: 0.397000; val_acc: 0.341000
(Iteration 16201 / 24500) loss: 1.010902
(Iteration 16301 / 24500) loss: 0.900927
(Iteration 16401 / 24500) loss: 1.019854
(Iteration 16501 / 24500) loss: 0.849295
(Iteration 16601 / 24500) loss: 0.822497
(Epoch 34 / 50) train acc: 0.413000; val_acc: 0.360000
(Iteration 16701 / 24500) loss: 0.923418
(Iteration 16801 / 24500) loss: 1.166213
(Iteration 16901 / 24500) loss: 1.081067
(Iteration 17001 / 24500) loss: 1.001114
(Iteration 17101 / 24500) loss: 0.804655
(Epoch 35 / 50) train acc: 0.398000; val_acc: 0.338000
(Iteration 17201 / 24500) loss: 0.897361
(Iteration 17301 / 24500) loss: 0.975102
(Iteration 17401 / 24500) loss: 1.136152
(Iteration 17501 / 24500) loss: 0.915982
(Iteration 17601 / 24500) loss: 0.917310
(Epoch 36 / 50) train acc: 0.402000; val_acc: 0.353000
(Iteration 17701 / 24500) loss: 1.035273
(Iteration 17801 / 24500) loss: 0.918930
(Iteration 17901 / 24500) loss: 0.925212
(Iteration 18001 / 24500) loss: 0.947147
(Iteration 18101 / 24500) loss: 0.906933
(Epoch 37 / 50) train acc: 0.394000; val_acc: 0.321000
(Iteration 18201 / 24500) loss: 0.781068
(Iteration 18301 / 24500) loss: 0.951483
(Iteration 18401 / 24500) loss: 0.858746
(Iteration 18501 / 24500) loss: 1.042784
(Iteration 18601 / 24500) loss: 1.044971
(Epoch 38 / 50) train acc: 0.383000; val_acc: 0.320000
(Iteration 18701 / 24500) loss: 0.835460
(Iteration 18801 / 24500) loss: 0.720850
(Iteration 18901 / 24500) loss: 0.930591
(Iteration 19001 / 24500) loss: 0.857013
(Iteration 19101 / 24500) loss: 0.900942
(Epoch 39 / 50) train acc: 0.394000; val_acc: 0.325000
(Iteration 19201 / 24500) loss: 0.976922
(Iteration 19301 / 24500) loss: 0.934433
(Iteration 19401 / 24500) loss: 0.971827
(Iteration 19501 / 24500) loss: 0.763441
(Epoch 40 / 50) train acc: 0.386000; val_acc: 0.341000
(Iteration 19601 / 24500) loss: 1.001094
(Iteration 19701 / 24500) loss: 0.946108
(Iteration 19801 / 24500) loss: 1.071578
(Iteration 19901 / 24500) loss: 0.907535
(Iteration 20001 / 24500) loss: 0.953595
(Epoch 41 / 50) train acc: 0.410000; val_acc: 0.345000
(Iteration 20101 / 24500) loss: 1.030167
(Iteration 20201 / 24500) loss: 0.916112
(Iteration 20301 / 24500) loss: 1.004554
(Iteration 20401 / 24500) loss: 0.911879
(Iteration 20501 / 24500) loss: 0.796875
(Epoch 42 / 50) train acc: 0.385000; val_acc: 0.354000
(Iteration 20601 / 24500) loss: 0.931571
(Iteration 20701 / 24500) loss: 0.872674
(Iteration 20801 / 24500) loss: 0.922485
(Iteration 20901 / 24500) loss: 0.769589
(Iteration 21001 / 24500) loss: 0.810918
(Epoch 43 / 50) train acc: 0.372000; val_acc: 0.347000
(Iteration 21101 / 24500) loss: 0.939233
(Iteration 21201 / 24500) loss: 0.825838
(Iteration 21301 / 24500) loss: 1.094397
(Iteration 21401 / 24500) loss: 0.901440
(Iteration 21501 / 24500) loss: 0.945438
(Epoch 44 / 50) train acc: 0.386000; val_acc: 0.335000
(Iteration 21601 / 24500) loss: 1.022207
(Iteration 21701 / 24500) loss: 0.781496
(Iteration 21801 / 24500) loss: 1.110713
(Iteration 21901 / 24500) loss: 0.916044
(Iteration 22001 / 24500) loss: 0.837923
(Epoch 45 / 50) train acc: 0.363000; val_acc: 0.350000
(Iteration 22101 / 24500) loss: 0.815166
(Iteration 22201 / 24500) loss: 1.094895
(Iteration 22301 / 24500) loss: 0.793580
(Iteration 22401 / 24500) loss: 0.869294
(Iteration 22501 / 24500) loss: 0.902156
(Epoch 46 / 50) train acc: 0.381000; val_acc: 0.343000
(Iteration 22601 / 24500) loss: 0.919967
(Iteration 22701 / 24500) loss: 0.786360
(Iteration 22801 / 24500) loss: 0.980892
(Iteration 22901 / 24500) loss: 0.758517
(Iteration 23001 / 24500) loss: 0.794294
(Epoch 47 / 50) train acc: 0.435000; val_acc: 0.349000
(Iteration 23101 / 24500) loss: 1.124971
(Iteration 23201 / 24500) loss: 0.853954
(Iteration 23301 / 24500) loss: 0.803810
(Iteration 23401 / 24500) loss: 1.076629
(Iteration 23501 / 24500) loss: 0.783814
(Epoch 48 / 50) train acc: 0.419000; val_acc: 0.333000
(Iteration 23601 / 24500) loss: 1.013780
(Iteration 23701 / 24500) loss: 1.006097
(Iteration 23801 / 24500) loss: 0.861510
(Iteration 23901 / 24500) loss: 0.957869
(Iteration 24001 / 24500) loss: 0.941365
(Epoch 49 / 50) train acc: 0.405000; val_acc: 0.358000
(Iteration 24101 / 24500) loss: 0.827097
(Iteration 24201 / 24500) loss: 0.927091
(Iteration 24301 / 24500) loss: 0.820969
(Iteration 24401 / 24500) loss: 0.901925
(Epoch 50 / 50) train acc: 0.412000; val_acc: 0.358000
```

In [ ]:

1. Optimizer

a.

```python
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.
    # ================================================================ #
    v = (config['momentum'] * v) - (config['learning_rate'] * dw)
    next_w = w + v
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    config['velocity'] = v

    return next_w, config
```

b.

```python
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.
    # ================================================================ #
    v_old = v
    v = config['momentum']*v_old - config['learning_rate']*dw
    next_w = w + v + config['momentum']*(v - v_old)

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

    config['velocity'] = v

    return next_w, config
```

```python
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.
    # ================================================================ #
    config['a'] = config['decay_rate'] * config['a'] + (1-config['decay_rate']) * np.multiply(dw, dw)
    next_w = w - np.multiply(config['learning_rate'] / (np.sqrt(config['a'] + config['epsilon'])), dw)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config
```

c.

2. Batch Normalization

   a. Batch Forward

```python
    minibatch_mean = np.mean(x, axis=0)
    minibatch_var = np.var(x, axis=0)
    x_normalize = (x - minibatch_mean) / np.sqrt(minibatch_var + eps)
    out = gamma * x_normalize + beta

    running_mean = momentum * running_mean + (1 - momentum) * minibatch_mean
    running_var = momentum * running_var + (1 - momentum) * minibatch_var
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    cache = {
      'minibatch_var': minibatch_var,
      'x_centralize': (x - minibatch_mean),
      'x_normalize': x_normalize,
      'gamma': gamma,
      'eps': eps
    }
    # ================================================================ #
    # 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'.
    # ================================================================ #

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

i.

   b. Batch Backward:

```python
"""
dx, dgamma, dbeta = None, None, None

# ================================================================ #
# YOUR CODE HERE:
#   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
# ================================================================ #
N = dout.shape[0]
minibatch_var = cache.get('minibatch_var')
x_centralize = cache.get('x_centralize')
x_normalize = cache.get('x_normalize')
gamma = cache.get('gamma')
eps = cache.get('eps')

# calculate dx
dxhat = dout * gamma
dxmu1 = dxhat / np.sqrt(minibatch_var + eps)
sqrt_var = np.sqrt(minibatch_var + eps)
dsqrt_var = -np.sum(dxhat * x_centralize, axis=0) / (sqrt_var**2)
dvar = dsqrt_var * 0.5 / sqrt_var
dxmu2 = 2 * x_centralize * dvar * np.ones_like(dout) / N
dx1 = dxmu1 + dxmu2
dx2 = -np.sum(dx1, axis=0) * np.ones_like(dout) / N
dx = dx1 + dx2

# calculate dbeta and dgamma
dbeta = np.sum(dout, axis=0)
dgamma = np.sum(dout * x_normalize, axis=0)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return dx, dgamma, dbeta
```

i.

3. Dropout
   a. Dropout forward:

```python
out = None

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

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

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

elif mode == 'test':

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

  mask = (np.random.random_sample(x.shape) >= p) / (1 - p)
  out = x * mask

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

cache = (dropout_param, mask)
```

i.

   b. Dropout Backward:

```
    """
    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

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

i.