Batch Normalization

In this notebook, you will implement the batch normalization layers of a neural network to increase its performance. If you have any confusion, 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).

In [288]:
```
## 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_numeri
cal_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plo
ts
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
```

In [289]:
```
# 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 [290]:
```
# 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:  [ -4.77273782  29.43839171  13.98384247]
  stds:  [ 26.35667822  29.20543313  32.15623214]
After batch normalization (gamma=1, beta=0)
  mean:  [ -3.38618023e-17   6.66133815e-18  -7.54951657e-17]
  std:  [ 0.99999999  0.99999999  1.        ]
After batch normalization (nontrivial gamma, beta)
  means:  [ 11.  12.  13.]
  stds:  [ 0.99999999  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 [291]:
```
# 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:  [ 0.08593135 -0.03563948 -0.05269799]
  stds:  [ 1.01983728  0.98978012  1.0656028 ]
```

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 [292]:
```
# 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, dgamma, dbeta)

print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
[[ -1.08393230e-01  -7.25486547e-03   6.54049767e-02   8.13297213e-0
2
   -6.77128463e-02]
 [  1.49957469e-02  -2.06453845e-02  -6.25766188e-02   4.64155138e-0
2
    1.35075912e-02]
 [ -2.16937699e-04   3.23538320e-02   6.35193145e-02   1.40816420e-0
1
    1.75741579e-01]
 [  9.36144210e-02  -4.45358205e-03  -6.63476725e-02  -2.68561655e-0
1
   -1.21536324e-01]] [ 2.46074274  1.17074281 -1.50157428 -0.5387052
5  0.58101496] [ 1.47707902 -0.19294516  3.33409901  0.8237985  -0.6
5006074]
dx error:  5.9114822945e-09
dgamma error:  3.47530270348e-12
dbeta error:  3.27572911404e-12
```

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 [293]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

print(X.shape)

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.float
64,
                            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, gra
ds[name])))
  if reg == 0: print('\n')
```

```
(2, 15)
Running check with reg =  0
Initial loss:  2.18570087163
W1 relative error: 0.0017620961330047724
W2 relative error: 1.5155245951274783e-05
W3 relative error: 4.789600547411837e-07
b1 relative error: 6.938893903907228e-09
b2 relative error: 6.661338147750939e-08
b3 relative error: 4.537302024147189e-07
beta1 relative error: 6.149690539945989e-07
beta2 relative error: 4.795865813281534e-07
gamma1 relative error: 6.210851395420151e-07
gamma2 relative error: 4.785530895915207e-07


Running check with reg =  3.14
Initial loss:  6.93152340975
W1 relative error: 2.124938522100045e-06
W2 relative error: 3.340676923521693e-05
W3 relative error: 6.237179420256357e-05
b1 relative error: 1.6653345369377348e-08
b2 relative error: 1.2212453270876722e-07
b3 relative error: 4.1214720252785485e-07
beta1 relative error: 5.051135620034601e-07
beta2 relative error: 4.374539805719088e-07
gamma1 relative error: 5.0293038836873e-07
gamma2 relative error: 4.37858551157119e-07
```

Training a deep fully connected network with batch normalization.

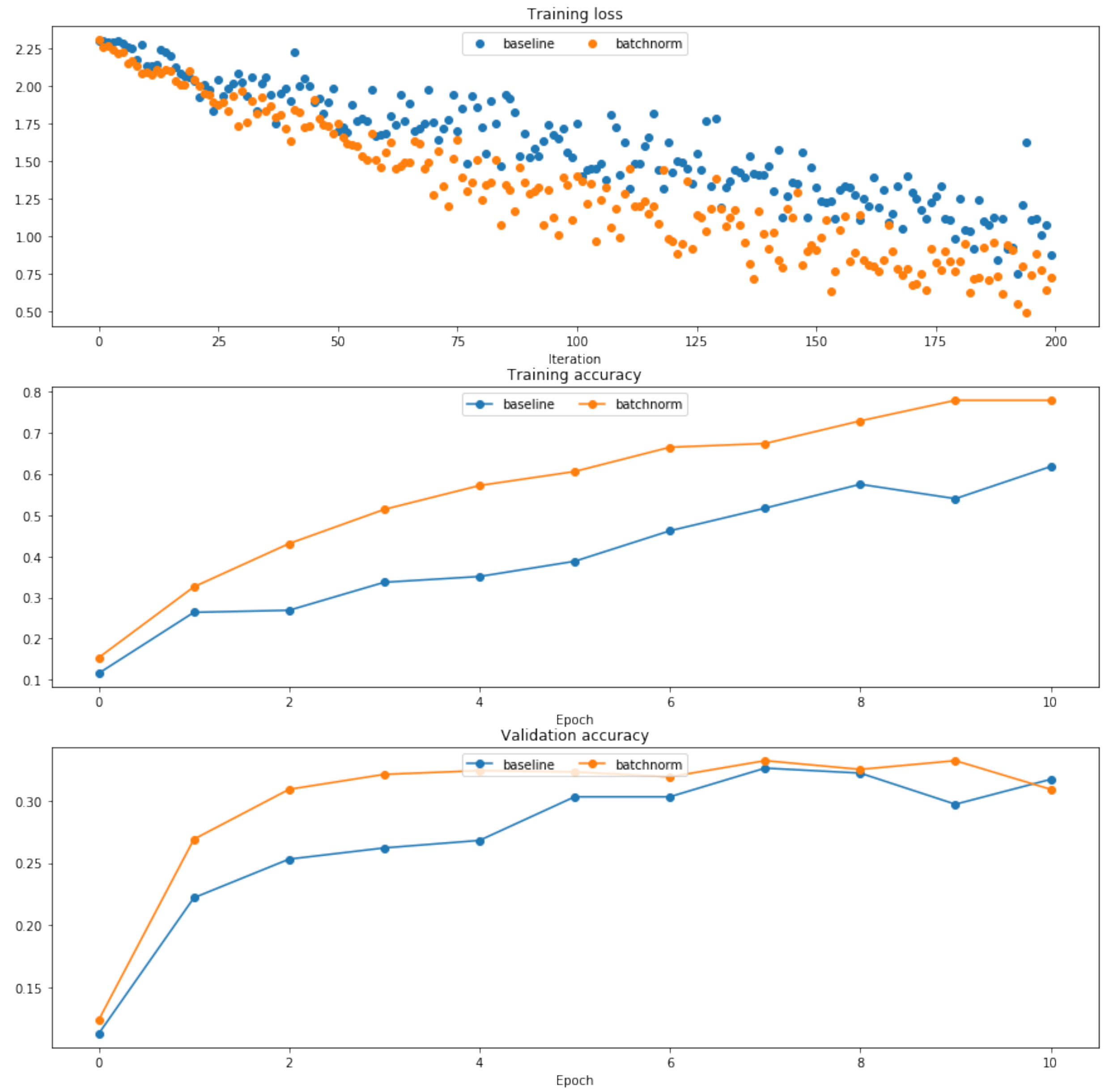
To see if batchnorm helps, let's train a deep neural network with and without batch normalization.

In [294]:
```
# 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, u
se_batchnorm=True)
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_
batchnorm=False)

print(small_data['X_train'].shape)
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()
```

```
(1000, 3, 32, 32)
(Iteration 1 / 200) loss: 2.309796
(Epoch 0 / 10) train acc: 0.154000; val_acc: 0.124000
(Epoch 1 / 10) train acc: 0.326000; val_acc: 0.269000
(Epoch 2 / 10) train acc: 0.431000; val_acc: 0.309000
(Epoch 3 / 10) train acc: 0.514000; val_acc: 0.321000
(Epoch 4 / 10) train acc: 0.572000; val_acc: 0.324000
(Epoch 5 / 10) train acc: 0.606000; val_acc: 0.323000
(Epoch 6 / 10) train acc: 0.665000; val_acc: 0.319000
(Epoch 7 / 10) train acc: 0.674000; val_acc: 0.332000
(Epoch 8 / 10) train acc: 0.729000; val_acc: 0.325000
(Epoch 9 / 10) train acc: 0.779000; val_acc: 0.332000
(Epoch 10 / 10) train acc: 0.779000; val_acc: 0.309000
(Iteration 1 / 200) loss: 2.303343
(Epoch 0 / 10) train acc: 0.116000; val_acc: 0.113000
(Epoch 1 / 10) train acc: 0.264000; val_acc: 0.222000
(Epoch 2 / 10) train acc: 0.269000; val_acc: 0.253000
(Epoch 3 / 10) train acc: 0.337000; val_acc: 0.262000
(Epoch 4 / 10) train acc: 0.351000; val_acc: 0.268000
(Epoch 5 / 10) train acc: 0.388000; val_acc: 0.303000
(Epoch 6 / 10) train acc: 0.462000; val_acc: 0.303000
(Epoch 7 / 10) train acc: 0.517000; val_acc: 0.326000
(Epoch 8 / 10) train acc: 0.575000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.540000; val_acc: 0.297000
(Epoch 10 / 10) train acc: 0.618000; val_acc: 0.317000
```

In [295]:
```
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()
```

```
/Users/Jonny/anaconda3/lib/python3.6/site-packages/matplotlib/cbook/
deprecation.py:106: MatplotlibDeprecationWarning: Adding an axes usi
ng the same arguments as a previous axes currently reuses the earlie
r instance.  In a future version, a new instance will always be crea
ted and returned.  Meanwhile, this warning can be suppressed, and th
e future behavior ensured, by passing a unique label to each axes in
stance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



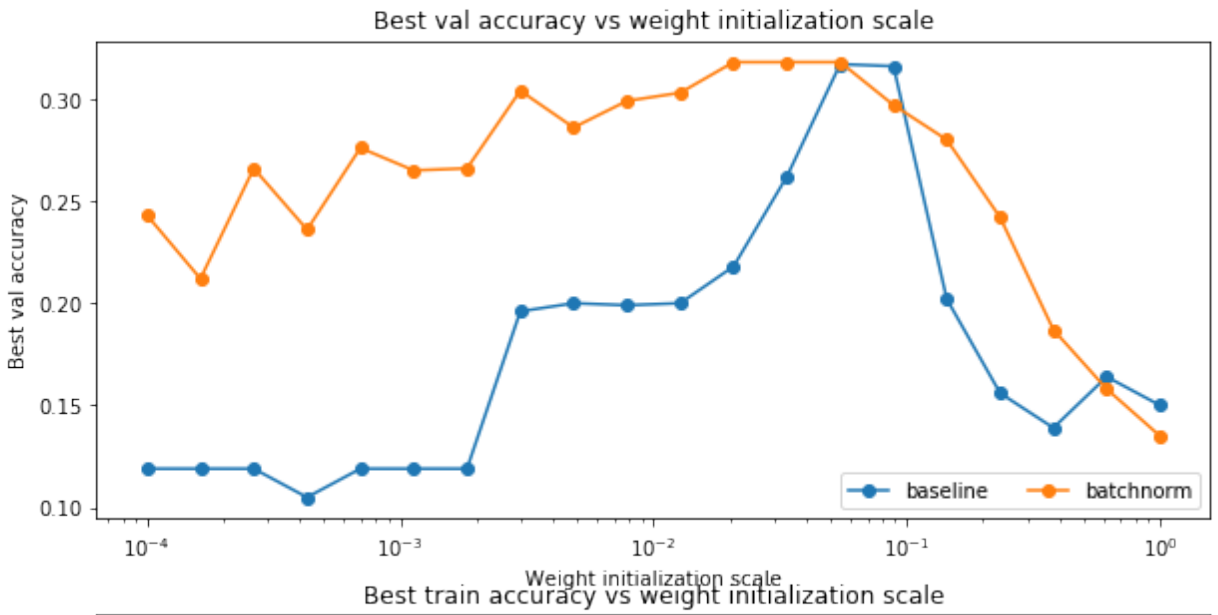
Batchnorm and initialization

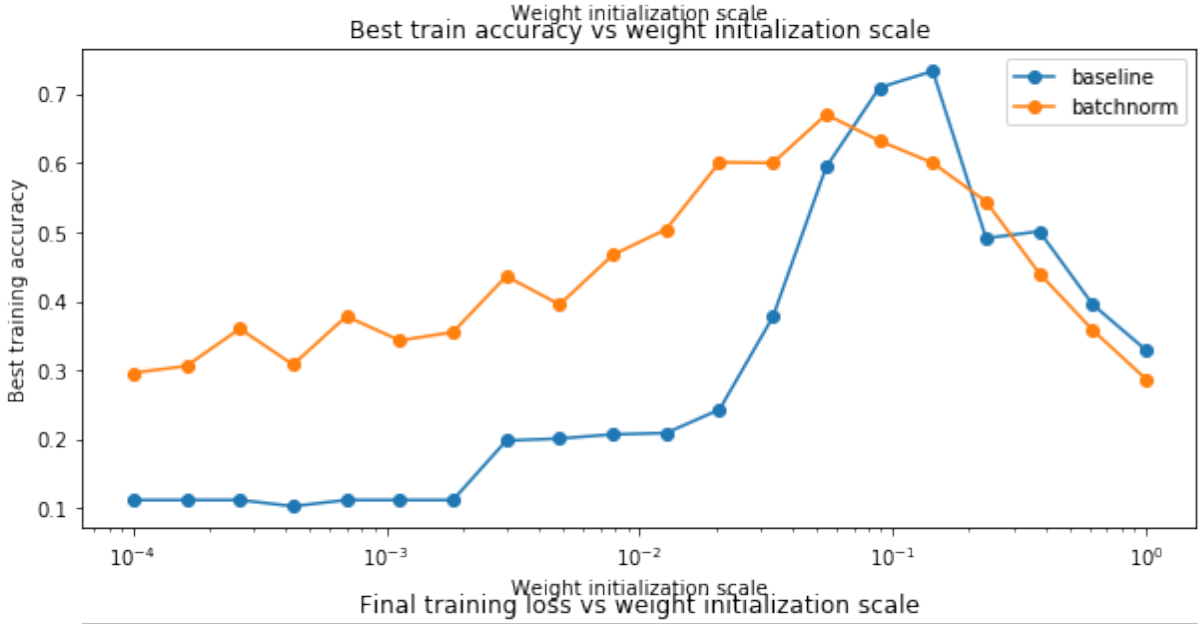
The following cells run an experiment where for a deep network, the initialization is varied. We do training for when batchnorm layers are and are not included.

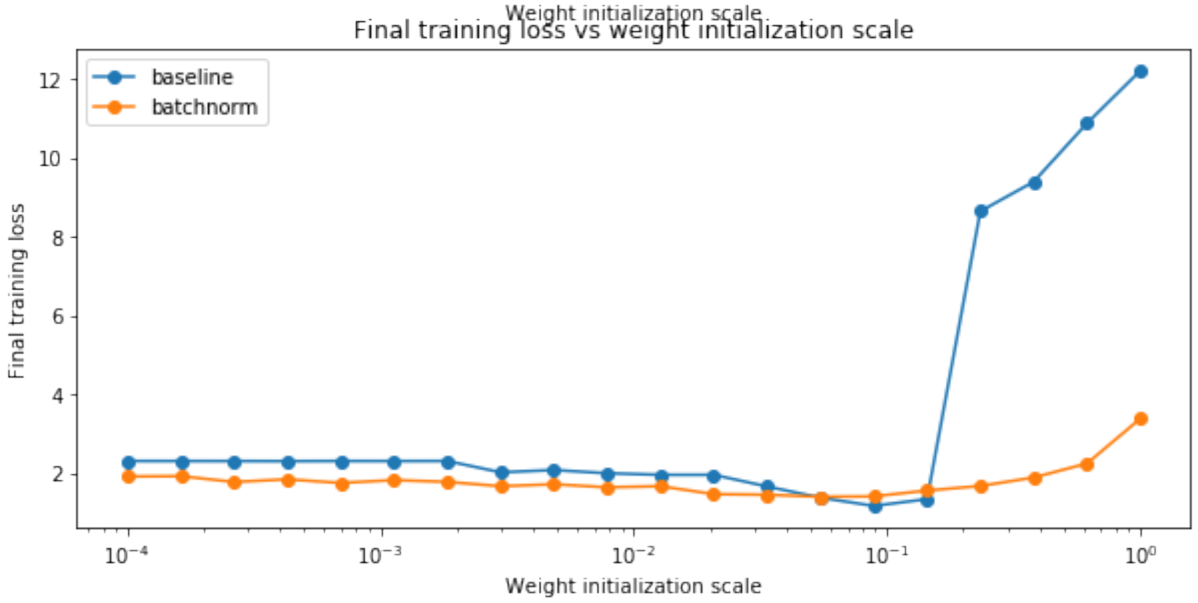
In [296]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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, us
e_batchnorm=False)

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

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

```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

In [297]:
```
# 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='batchnor
m')
plt.legend()

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


Best val accuracy vs weight initialization scale
Best val accuracy
Weight initialization scale
baseline
batchnorm


Best train accuracy vs weight initialization scale
Best training accuracy
Weight initialization scale
baseline
batchnorm


Final training loss vs weight initialization scale
Final training loss
Weight initialization scale
baseline
batchnorm

Question:

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

Answer:

These experiments show that the deep network with batch normalization is less sensitive to weight initializations. In the first experiment, the accuracy for the regular network is extremely low for low weight initialization standard deviations. While an optimal weight initialization is achievable (near 10^-1), this would require a hyperparameter sweep. By normalizing the data in each mini-batch, we are esentially making each layer independent in terms of mean and variance. The regular method (no batch normalization) is heavily dependent on weight initializations because these affect the mean and variance of the outputs as data propagates through the network.

The hypothesis is that a batch normalized network will exhibit a much smoother accuracy vs. weight initialization curve vs. the regular network. This is confirmed in all three of the figures. In the last figure, with large stddev for weight initializations, the training loss for the regular network explodes whereas the normalized network's loss increases but does not shoot up. All curves are smoother, confirming our hypothesis.