# 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 a solid API for building these modular frameworks and training them, and we will use this very well implemented framework as opposed to "reinventing the wheel." This includes using the Solver, various utility functions, and the 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_gradi
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-ipyth
%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 [27]:
```
# 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:  [-12.18161056   7.37672575 -22.74890196]
  stds:  [32.30017774 29.11000724 36.60663825]
After batch normalization (gamma=1, beta=0)
  mean:  [ 3.60822483e-17 -1.42247325e-16 -4.66293670e-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 [30]:
```
# 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:  [ 0.00467564 -0.03482305  0.06476255]
  stds:  [0.92113549 1.01524581 0.95984865]
```

## 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 [31]:
```
# 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 gamma: batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda beta: 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:  1.1160370604910315e-09
dgamma error:  1.0607692151563754e-11
dbeta error:  5.363917926020398e-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 [49]:
```
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=Fals
        print('{} relative error: {}'.format(name, rel_error(grad_num, grads[n
    if reg == 0: print('\n')
```

```
Running check with reg =  0
Initial loss:  2.2586867547771705
W1 relative error: 5.442561142286271e-05
W2 relative error: 9.14430686528561e-06
W3 relative error: 3.6599618127959146e-10
b1 relative error: 2.1649348980190553e-07
b2 relative error: 3.2862601528904634e-06
b3 relative error: 1.2554198907223018e-10
beta1 relative error: 3.978576318366779e-08
beta2 relative error: 7.258674515555404e-08
gamma1 relative error: 4.672895789641736e-08
gamma2 relative error: 1.3085650060962925e-08


Running check with reg =  3.14
Initial loss:  6.654592794005303
W1 relative error: 5.655201970615687e-06
W2 relative error: 8.968138637930617e-07
W3 relative error: 1.7406553023338138e-08
b1 relative error: 7.216449660063518e-07
b2 relative error: 5.151434834260726e-06
b3 relative error: 2.9279858223819383e-10
beta1 relative error: 1.3056598786124765e-08
beta2 relative error: 1.1416562097302202e-08
gamma1 relative error: 1.3226556475182594e-08
gamma2 relative error: 1.57708038149929e-08
```

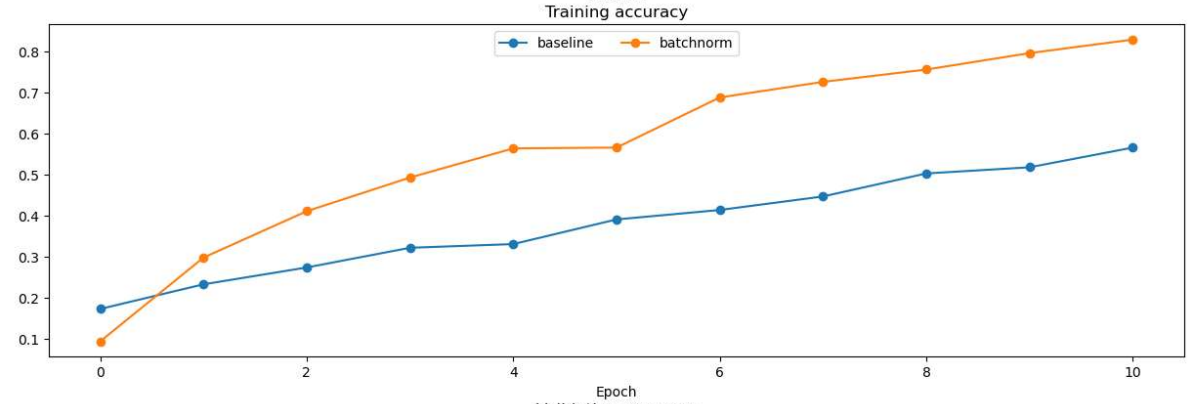
# Training a deep fully connected network with batch normalization.

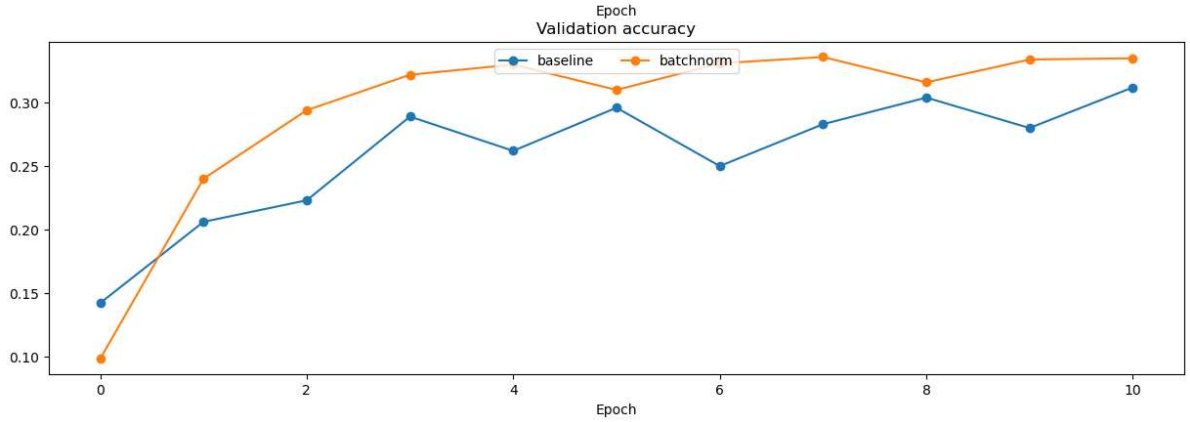
To see if batchnorm helps, let's train a deep neural network with and without batch normalization.

In [38]:
```
# 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_batchn
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm

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: 2.315769
(Epoch 0 / 10) train acc: 0.093000; val_acc: 0.098000
(Epoch 1 / 10) train acc: 0.297000; val_acc: 0.240000
(Epoch 2 / 10) train acc: 0.410000; val_acc: 0.294000
(Epoch 3 / 10) train acc: 0.492000; val_acc: 0.322000
(Epoch 4 / 10) train acc: 0.563000; val_acc: 0.330000
(Epoch 5 / 10) train acc: 0.565000; val_acc: 0.310000
(Epoch 6 / 10) train acc: 0.687000; val_acc: 0.331000
(Epoch 7 / 10) train acc: 0.725000; val_acc: 0.336000
(Epoch 8 / 10) train acc: 0.755000; val_acc: 0.316000
(Epoch 9 / 10) train acc: 0.795000; val_acc: 0.334000
(Epoch 10 / 10) train acc: 0.828000; val_acc: 0.335000
(Iteration 1 / 200) loss: 2.301997
(Epoch 0 / 10) train acc: 0.172000; val_acc: 0.142000
(Epoch 1 / 10) train acc: 0.232000; val_acc: 0.206000
(Epoch 2 / 10) train acc: 0.273000; val_acc: 0.223000
(Epoch 3 / 10) train acc: 0.321000; val_acc: 0.289000
(Epoch 4 / 10) train acc: 0.330000; val_acc: 0.262000
(Epoch 5 / 10) train acc: 0.390000; val_acc: 0.296000
(Epoch 6 / 10) train acc: 0.413000; val_acc: 0.250000
(Epoch 7 / 10) train acc: 0.446000; val_acc: 0.283000
(Epoch 8 / 10) train acc: 0.502000; val_acc: 0.304000
(Epoch 9 / 10) train acc: 0.517000; val_acc: 0.280000
(Epoch 10 / 10) train acc: 0.565000; val_acc: 0.312000
```

In [39]:
```
fig, axes = plt.subplots(3, 1)

ax = axes[0]
ax.set_title('Training loss')
ax.set_xlabel('Iteration')

ax = axes[1]
ax.set_title('Training accuracy')
ax.set_xlabel('Epoch')

ax = axes[2]
ax.set_title('Validation accuracy')
ax.set_xlabel('Epoch')

ax = axes[0]
ax.plot(solver.loss_history, 'o', label='baseline')
ax.plot(bn_solver.loss_history, 'o', label='batchnorm')

ax = axes[1]
ax.plot(solver.train_acc_history, '-o', label='baseline')
ax.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

ax = axes[2]
ax.plot(solver.val_acc_history, '-o', label='baseline')
ax.plot(bn_solver.val_acc_history, '-o', label='batchnorm')

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

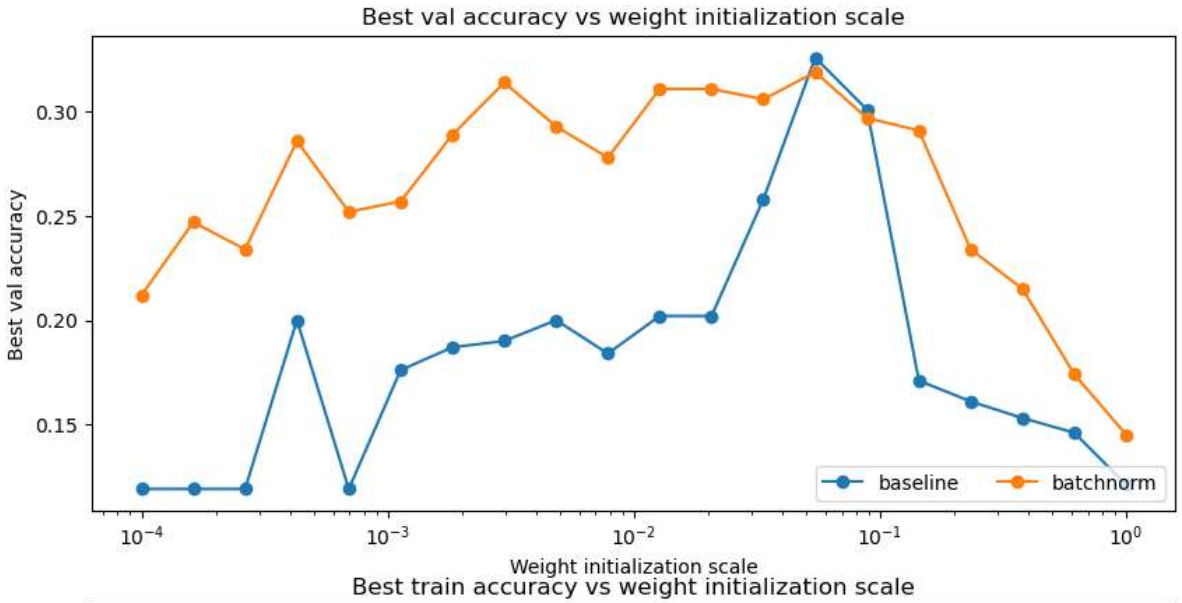

# Batchnorm and initialization

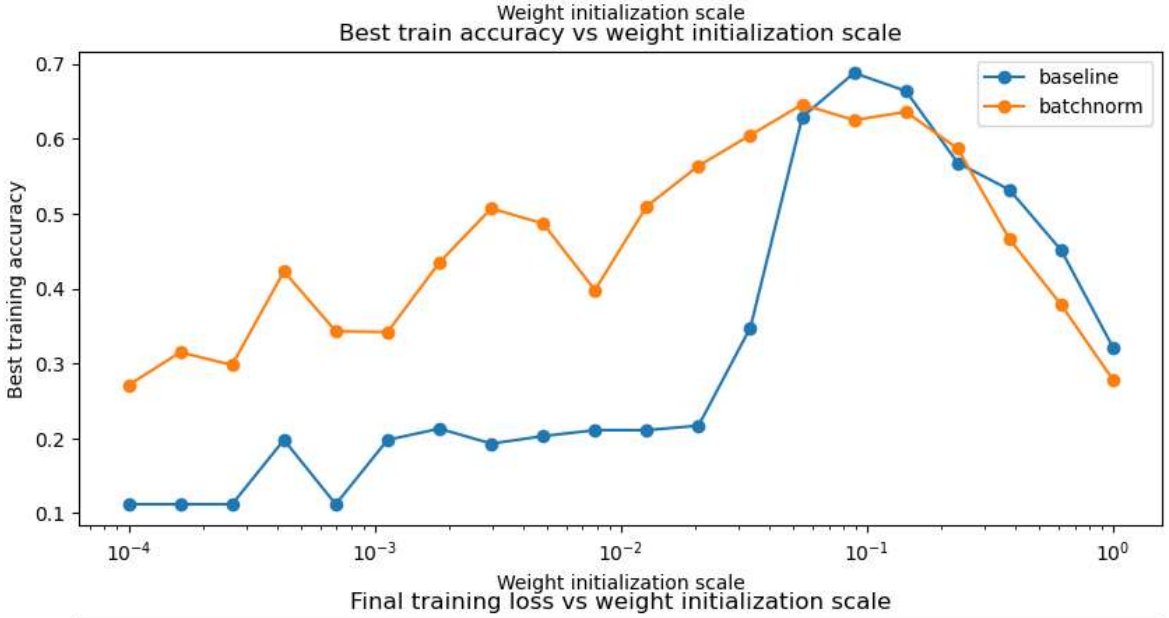
The following cells run an experiment where for a deep network, the initialization is varied. We do training for when batchnorm layers are and are not included.

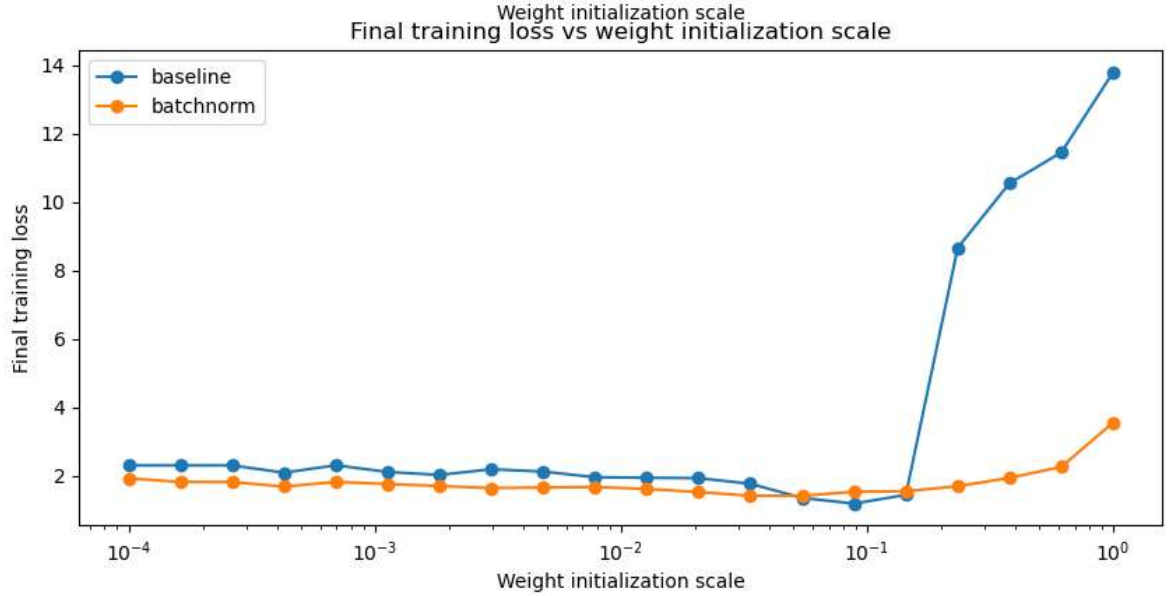
In [45]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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_ba
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batch

    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 [46]:
```
# 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
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:

The findings are that batch normalization achieves higher accuracies (both training and validation) for the most part. When the weight initialization scale is less than 10^-1, models with batch norm have higher accuracies than those without batch norm; the accuracy for the baseline model for small weight scales is very low. The baseline model only scores well (either equal to or higher than the batch norm model) when the weight initialization scale is 10^-1, while the batch norm model scores better than the baseline model for the most part and only decreases in accuracy as the weight initialization scale approches 10^0. We can see best that the batch norm model is less sensitive to weight initializations in the training loss graph, where the batchnorm loss is very smooth regardless of the initial weights (and only increases slightly), while the loss for the baseline model explodes at initial weight scale > 10^-1.

This is because the batch norm model is less sensitive to hyperparameters like weight initializations. Because we normalize the the output of each layer to have unit statistics in batch norm, the parameters in lower layers do not change the statistics of the input to the next layer. In the baseline model, the distribution of inputs to each layer changes as learning occurs in previous layers because each layer's statistics depend on previous layers' statistics. The baseline model is heavily dependent on weight initialization because these affect the means and variances of each layer.

In [ ]: