batch_normalization

March 12, 2019

1 Batch Normalization

In this task, we implement batch normalization, which normalizes hidden layers and makes the training procedure more stable.

Reference: Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

In [1]:
```
# As usual, a bit of setup
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from data_utils import get_CIFAR10_data
from implementations.layers import batchnorm_forward

%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))))

def print_mean_std(x,axis=0):
    print('  means: ', x.mean(axis=axis))
    print('  stds:  ', x.std(axis=axis))
    print()
```

In [2]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
```

```
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

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

1.1 Batch normalization: forward

In the file `implementations/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above would be helpful!

In [3]:
```
# A very simple example

xtrain = np.array([[10], [20], [30]])

xtest = np.array([[25]])


# initialize parameters for batch normalization
bn_param = {}
bn_param['mode'] = 'train'
bn_param['eps'] = 1e-4
bn_param['momentum'] = 0.95

# initialize the running mean as zero and the running variance as one
bn_param['running_mean'] = np.zeros([1, xtrain.shape[1]])
bn_param['running_var'] = np.ones([1, xtrain.shape[1]])

# gamma and beta do not make changes to the standardization result from the first step
gamma = np.ones([1])
beta = np.zeros([1])

print('Before batch normalization, xtrain has ')
print_mean_std(xtrain,axis=0)

xnorm = batchnorm_forward(xtrain, gamma, beta, bn_param)

print('After batch normalization, xtrain has ')
print_mean_std(xnorm,axis=0) # The mean and std should be 0 and 1 respectively
```

```
    print('After batch normalization, the running mean and the running variance are updated
    print(bn_param['running_mean']) # should be 1.0
    print(bn_param['running_var']) # should be 4.283


    for iter in range(1000):
        xnorm = batchnorm_forward(xtrain, gamma, beta, bn_param)

    print('After many iterations, the running mean and the running variance are updated to
    print(bn_param['running_mean']) # should be 20, the mean of xtrain
    print(bn_param['running_var']) # should be 66.667, the variance of xtrain


    # enter test mode,
    bn_param['mode'] = 'test'
    xtest_norm = batchnorm_forward(xtest, gamma, beta, bn_param)

    print('Before batch normalization, xtest becomes ') # should be [[0.61237198]]
    print(xtest_norm)
```

```
Before batch normalization, xtrain has
  means:  [20.]
  stds:   [8.16496581]

After batch normalization, xtrain has
  means:  [0.]
  stds:   [1.]

After batch normalization, the running mean and the running variance are updated to
[[1.]]
[[4.28333333]]
After many iterations, the running mean and the running variance are updated to
[[20.]]
[[66.66666667]]
Before batch normalization, xtest becomes
[[0.61237244]]
```

In [5]:
```
# Compare with tf.layers.batch_normalization

# Simulate the forward pass for a two-layer network
np.random.seed(15009)
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)
```

```python
# initialize parameters for batch normalization
bn_param = {}
bn_param['mode'] = 'train'
bn_param['eps'] = 1e-4
bn_param['momentum'] = 0.95
bn_param['running_mean'] = np.zeros([1, a.shape[1]])
bn_param['running_var'] = np.ones([1, a.shape[1]])

# random gamma and beta
gamma = np.random.rand(D3) + 1.0
beta = np.random.rand(D3)


# Setting up a tensorflow bn layer using the same set of parameters.

tf.reset_default_graph()
tfa = tf.placeholder(tf.float32, shape=[None, a.shape[1]])

# used to control the mode
is_training = tf.placeholder_with_default(False, (), 'is_training')

# the axis setting is a little strange to me. But you can understand it as that the ax
# the running mean
tfa_norm = tf.layers.batch_normalization(tfa, axis=1, momentum=0.95, epsilon=bn_param[
                                         beta_initializer=tf.constant_initializer(beta
                                         gamma_initializer=tf.constant_initializer(gamm
                                         moving_mean_initializer=tf.zeros_initializer(
                                         moving_variance_initializer=tf.ones_initialize
                                         training=is_training)

# this operation is for undating running mean and running variance.
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

session = tf.Session()

# initialize parameters
session.run(tf.global_variables_initializer())


outputs = []
nbatch = 3
batch_size = 10

for ibatch in range(nbatch):

    # fetch the batch
```

```
        a_batch = a[ibatch * batch_size : (ibatch + 1) * batch_size]

        # batch normalization with your implementation
        a_nprun = batchnorm_forward(a_batch, gamma, beta, bn_param)

        # batch normalization with the tensorflow layer. Also update the running mean and 
        a_tfrun, _ = session.run([tfa_norm, update_ops], feed_dict={tfa: a_batch.astype(np

        rel_error(a_nprun,a_tfrun)
        print("Training batch %d: difference from the two implementations is %f" % (ibatch

    rel_error(a_nprun,a_tfrun)

    # enterining test mode
    bn_param['mode'] = 'test'

    for ibatch in range(nbatch):
        a_batch = a[ibatch * batch_size : (ibatch + 1) * batch_size]

        a_nprun = batchnorm_forward(a_batch, gamma, beta, bn_param)

        # run batch normalization in test mode. No need to update the running mean and var
        a_tfrun = session.run(tfa_norm, feed_dict={tfa: a_batch.astype(np.float32)})


        print("Test batch %d: difference from the two implementations is %f" % (ibatch, rel
```

```
Training batch 0: difference from the two implementations is 0.000001
Training batch 1: difference from the two implementations is 0.000017
Training batch 2: difference from the two implementations is 0.000001
Test batch 0: difference from the two implementations is 0.000000
Test batch 1: difference from the two implementations is 0.000001
Test batch 2: difference from the two implementations is 0.000001
```

1.2 Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization. Then you need to go back to your `FullyConnectedNet` in the file `implementations/fc_net.py`. Modify the implementation to add batch normalization.

When the `use_bn` flag is set, the network should apply batch normalization before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized.

2 Batchnorm for deep networks

Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

```
In [43]: from implementations.fc_net import FullyConnectedNet

         np.random.seed(15009)
         # Try training a very deep net with batchnorm
         hidden_dims = [100, 100, 100, 100, 100]

         num_train = 1000

         X_train = data['X_train'][:num_train]
         X_train = np.reshape(X_train, [X_train.shape[0], -1])
         y_train = data['y_train'][:num_train]

         X_val = data['X_val']
         X_val = np.reshape(X_val, [X_val.shape[0], -1])
         y_val = data['y_val']


         bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                                      hidden_size=hidden_dims,
                                      output_size=10,
                                      centering_data=True,
                                      use_dropout=False,
                                      use_bn=True)

         # use an aggresive learning rate
         bn_trace = bn_model.train(X_train, y_train, X_val, y_val,
                                   learning_rate=5e-4,
                                   reg=np.float32(0.01),
                                   keep_prob=0.5,
                                   num_iters=800,
                                   batch_size=100,
                                   verbose=True) # train the model with batch normalization
```

```
iteration 0 / 800: objective 232.335617
iteration 100 / 800: objective 58.288155
iteration 200 / 800: objective 5.658176
iteration 300 / 800: objective 3.893709
iteration 400 / 800: objective 3.395476
iteration 500 / 800: objective 3.150912
iteration 600 / 800: objective 3.003807
iteration 700 / 800: objective 2.905491
```

Train a fully connected network without batch normalization

```
In [37]: model = FullyConnectedNet(input_size=X_train.shape[1],
                                   hidden_size=hidden_dims,
                                   output_size=10,
```

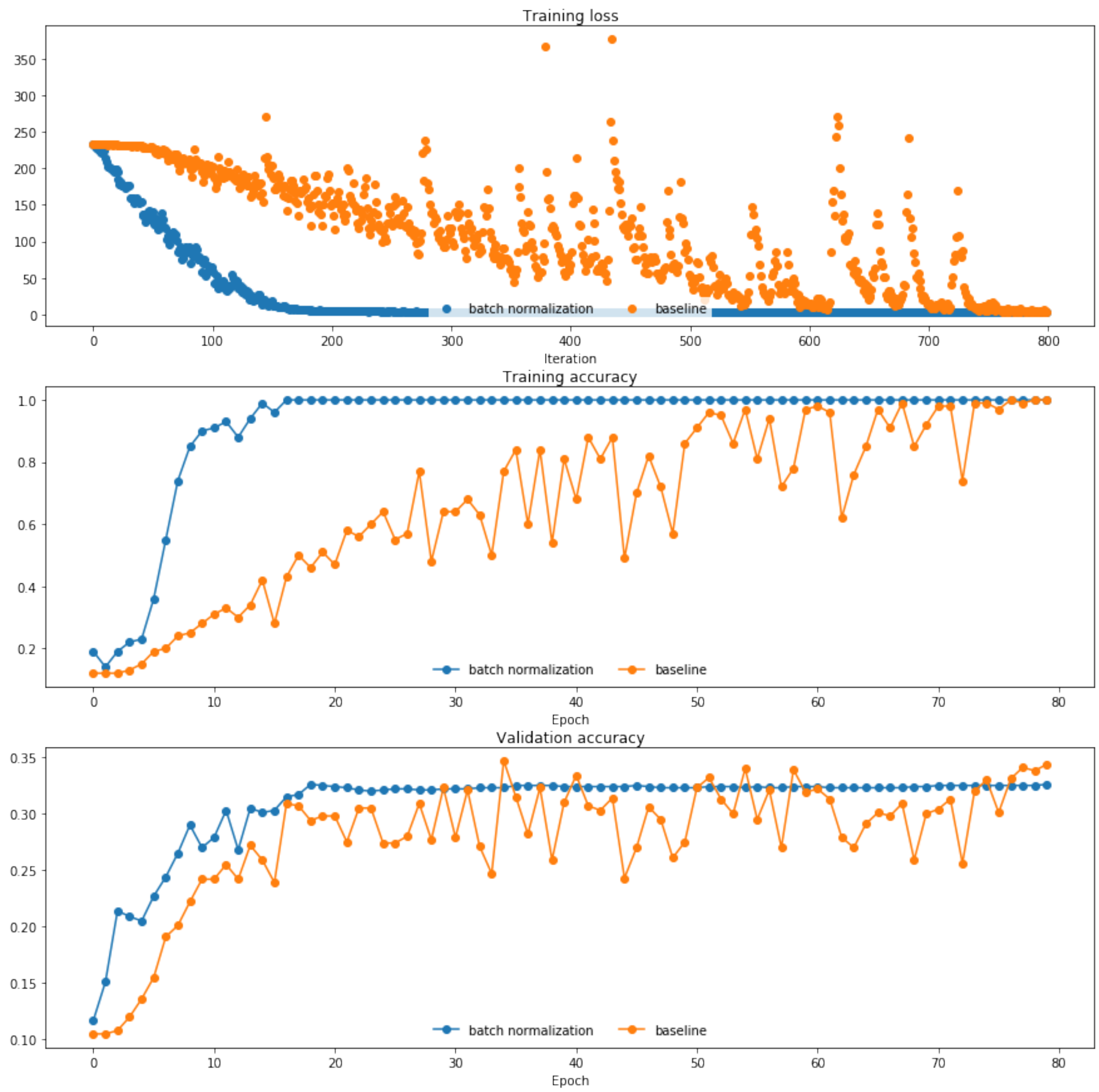
```
                                    centering_data=True,
                                    use_dropout=False,
                                    use_bn=False)

         # use an aggresive learning rate
         baseline_trace = model.train(X_train, y_train, X_val, y_val,
                                       learning_rate=5e-4,
                                       reg=np.float32(0.01),
                                       num_iters=800,
                                       batch_size=100,
                                       verbose=True) # train the model without batch normalizati
```

```
iteration 0 / 800: objective 232.689148
iteration 100 / 800: objective 202.023819
iteration 200 / 800: objective 153.215256
iteration 300 / 800: objective 117.816414
iteration 400 / 800: objective 68.232361
iteration 500 / 800: objective 51.377590
iteration 600 / 800: objective 13.974775
iteration 700 / 800: objective 14.563780
```

Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

```
In [44]: def plot_training_history(title, label, bl_plot, bn_plots, bl_marker='.', bn_marker='
             """utility function for plotting training history"""
             plt.title(title)
             plt.xlabel(label)
             num_bn = len(bn_plots)

             for i in range(num_bn):
                 label='batch normalization'
                 if labels is not None:
                     label += str(labels[i])
                 plt.plot(bn_plots[i], bn_marker, label=label)

             label='baseline'
             if labels is not None:
                 label += str(labels[0])

             plt.plot(bl_plot, bl_marker, label=label)
             plt.legend(loc='lower center', ncol=num_bn+1)


         plt.subplot(3, 1, 1)
         plot_training_history('Training loss','Iteration', baseline_trace['objective_history']
                               [bn_trace['objective_history']], bl_marker='o', bn_marker='o')
```

```
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', baseline_trace['train_acc_history']
                      [bn_trace['train_acc_history']], bl_marker='-o', bn_marker='-o'
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', baseline_trace['val_acc_history']
                      [bn_trace['val_acc_history']], bl_marker='-o', bn_marker='-o')

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



3 Batch normalization and initialization

We will now run a small experiment to study the interaction of batch normalization and weight initialization.

The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second cell will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

LIPING: I tried multiple configurations, but I did not find significant improvement from batch normalization. See if you can get clear improvement with your configurations.

In [45]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 10000

X_train = data['X_train'][:num_train]
X_train = np.reshape(X_train, [X_train.shape[0], -1])
y_train = data['y_train'][:num_train]

X_val = data['X_val']
X_val = np.reshape(X_val, [X_val.shape[0], -1])
y_val = data['y_val']

bn_net_ws = {}
baseline_ws = {}

weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale=%f at round %d / %d' % (weight_scale, i + 1, len(weight_


  bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                               hidden_size=hidden_dims,
                               output_size=10,
                               centering_data=True,
                               use_dropout=False,
                               use_bn=True)

  # use an aggresive learning rate
  bn_net_ws[weight_scale] = bn_model.train(X_train, y_train, X_val, y_val,
                            learning_rate=1e-2,
                            reg=np.float32(1e-5),
                            keep_prob=0.5,
                            num_iters=1000,
                            batch_size=100,
                            verbose=True) # train the model with batch normalization
```

```
        model = FullyConnectedNet(input_size=X_train.shape[1],
                                  hidden_size=hidden_dims,
                                  output_size=10,
                                  centering_data=True,
                                  use_dropout=False,
                                  use_bn=True)

        # use an aggresive learning rate
        baseline_ws[weight_scale] = model.train(X_train, y_train, X_val, y_val,
                                    learning_rate=1e-2,
                                    reg=np.float32(1e-5),
                                    num_iters=1000,
                                    batch_size=100,
                                    verbose=True)
```

```
Running weight scale=0.000100 at round 1 / 20
iteration 0 / 1000: objective 233.189850
iteration 100 / 1000: objective 213.902039
iteration 200 / 1000: objective 192.353149
iteration 300 / 1000: objective 188.296875
iteration 400 / 1000: objective 171.414932
iteration 500 / 1000: objective 168.465851
iteration 600 / 1000: objective 155.278107
iteration 700 / 1000: objective 162.533249
iteration 800 / 1000: objective 152.299194
iteration 900 / 1000: objective 146.888519
iteration 0 / 1000: objective 230.705170
iteration 100 / 1000: objective 216.462936
iteration 200 / 1000: objective 205.590530
iteration 300 / 1000: objective 202.161789
iteration 400 / 1000: objective 187.147522
iteration 500 / 1000: objective 179.888824
iteration 600 / 1000: objective 170.057724
iteration 700 / 1000: objective 160.396988
iteration 800 / 1000: objective 154.108765
iteration 900 / 1000: objective 154.590164
Running weight scale=0.000162 at round 2 / 20
iteration 0 / 1000: objective 231.216522
iteration 100 / 1000: objective 212.255798
iteration 200 / 1000: objective 197.244400
iteration 300 / 1000: objective 191.965805
iteration 400 / 1000: objective 188.645523
iteration 500 / 1000: objective 178.723724
iteration 600 / 1000: objective 175.702026
iteration 700 / 1000: objective 163.626587
iteration 800 / 1000: objective 172.005768
iteration 900 / 1000: objective 153.029510
iteration 0 / 1000: objective 234.763519
```

```
iteration 100 / 1000: objective 220.328064
iteration 200 / 1000: objective 204.021286
iteration 300 / 1000: objective 203.361923
iteration 400 / 1000: objective 198.491867
iteration 500 / 1000: objective 189.717041
iteration 600 / 1000: objective 176.782776
iteration 700 / 1000: objective 168.987885
iteration 800 / 1000: objective 162.173172
iteration 900 / 1000: objective 150.489304
Running weight scale=0.000264 at round 3 / 20
iteration 0 / 1000: objective 228.916122
iteration 100 / 1000: objective 207.738663
iteration 200 / 1000: objective 204.375305
iteration 300 / 1000: objective 197.079208
iteration 400 / 1000: objective 183.681717
iteration 500 / 1000: objective 172.850174
iteration 600 / 1000: objective 164.575470
iteration 700 / 1000: objective 160.083267
iteration 800 / 1000: objective 151.550583
iteration 900 / 1000: objective 158.793671
iteration 0 / 1000: objective 228.990723
iteration 100 / 1000: objective 207.268585
iteration 200 / 1000: objective 198.851501
iteration 300 / 1000: objective 189.377426
iteration 400 / 1000: objective 175.886795
iteration 500 / 1000: objective 165.658539
iteration 600 / 1000: objective 164.889679
iteration 700 / 1000: objective 157.337112
iteration 800 / 1000: objective 148.951782
iteration 900 / 1000: objective 146.643051
Running weight scale=0.000428 at round 4 / 20
iteration 0 / 1000: objective 230.535248
iteration 100 / 1000: objective 203.746460
iteration 200 / 1000: objective 201.049683
iteration 300 / 1000: objective 191.514801
iteration 400 / 1000: objective 179.799026
iteration 500 / 1000: objective 173.799911
iteration 600 / 1000: objective 160.822021
iteration 700 / 1000: objective 153.684814
iteration 800 / 1000: objective 154.788635
iteration 900 / 1000: objective 142.595169
iteration 0 / 1000: objective 230.348511
iteration 100 / 1000: objective 209.631287
iteration 200 / 1000: objective 201.167435
iteration 300 / 1000: objective 197.067184
iteration 400 / 1000: objective 179.924408
iteration 500 / 1000: objective 177.369156
iteration 600 / 1000: objective 167.211639
```

```
iteration 700 / 1000: objective 163.410370
iteration 800 / 1000: objective 153.729431
iteration 900 / 1000: objective 152.585617
Running weight scale=0.000695 at round 5 / 20
iteration 0 / 1000: objective 231.667969
iteration 100 / 1000: objective 207.419830
iteration 200 / 1000: objective 199.112885
iteration 300 / 1000: objective 188.868378
iteration 400 / 1000: objective 181.915741
iteration 500 / 1000: objective 163.339966
iteration 600 / 1000: objective 162.051163
iteration 700 / 1000: objective 163.451355
iteration 800 / 1000: objective 144.159866
iteration 900 / 1000: objective 142.653580
iteration 0 / 1000: objective 230.594971
iteration 100 / 1000: objective 221.211746
iteration 200 / 1000: objective 204.929932
iteration 300 / 1000: objective 208.391785
iteration 400 / 1000: objective 188.673782
iteration 500 / 1000: objective 184.410980
iteration 600 / 1000: objective 167.118530
iteration 700 / 1000: objective 159.920700
iteration 800 / 1000: objective 155.284393
iteration 900 / 1000: objective 146.527756
Running weight scale=0.001129 at round 6 / 20
iteration 0 / 1000: objective 231.056152
iteration 100 / 1000: objective 216.812057
iteration 200 / 1000: objective 202.870682
iteration 300 / 1000: objective 187.071091
iteration 400 / 1000: objective 184.236038
iteration 500 / 1000: objective 170.143478
iteration 600 / 1000: objective 169.752975
iteration 700 / 1000: objective 161.809982
iteration 800 / 1000: objective 151.256165
iteration 900 / 1000: objective 148.594254
iteration 0 / 1000: objective 230.562378
iteration 100 / 1000: objective 208.735977
iteration 200 / 1000: objective 191.704269
iteration 300 / 1000: objective 186.752487
iteration 400 / 1000: objective 177.769943
iteration 500 / 1000: objective 174.821808
iteration 600 / 1000: objective 161.963974
iteration 700 / 1000: objective 148.823746
iteration 800 / 1000: objective 154.638779
iteration 900 / 1000: objective 145.665527
Running weight scale=0.001833 at round 7 / 20
iteration 0 / 1000: objective 232.726440
iteration 100 / 1000: objective 219.746735
```

```
iteration 200 / 1000: objective 203.194962
iteration 300 / 1000: objective 189.779373
iteration 400 / 1000: objective 189.381882
iteration 500 / 1000: objective 166.994354
iteration 600 / 1000: objective 164.440811
iteration 700 / 1000: objective 155.717102
iteration 800 / 1000: objective 149.184067
iteration 900 / 1000: objective 145.403320
iteration 0 / 1000: objective 230.228165
iteration 100 / 1000: objective 215.052811
iteration 200 / 1000: objective 205.093521
iteration 300 / 1000: objective 195.500244
iteration 400 / 1000: objective 185.565933
iteration 500 / 1000: objective 175.824020
iteration 600 / 1000: objective 166.680038
iteration 700 / 1000: objective 162.525375
iteration 800 / 1000: objective 163.046341
iteration 900 / 1000: objective 155.347595
Running weight scale=0.002976 at round 8 / 20
iteration 0 / 1000: objective 231.871552
iteration 100 / 1000: objective 210.456558
iteration 200 / 1000: objective 202.028580
iteration 300 / 1000: objective 195.819656
iteration 400 / 1000: objective 184.135803
iteration 500 / 1000: objective 181.513474
iteration 600 / 1000: objective 172.636688
iteration 700 / 1000: objective 165.690811
iteration 800 / 1000: objective 155.996368
iteration 900 / 1000: objective 149.967896
iteration 0 / 1000: objective 230.618286
iteration 100 / 1000: objective 209.890701
iteration 200 / 1000: objective 202.198181
iteration 300 / 1000: objective 189.465256
iteration 400 / 1000: objective 176.307556
iteration 500 / 1000: objective 173.637619
iteration 600 / 1000: objective 164.667557
iteration 700 / 1000: objective 154.692612
iteration 800 / 1000: objective 163.234711
iteration 900 / 1000: objective 150.940216
Running weight scale=0.004833 at round 9 / 20
iteration 0 / 1000: objective 230.446899
iteration 100 / 1000: objective 201.892807
iteration 200 / 1000: objective 188.108917
iteration 300 / 1000: objective 181.996353
iteration 400 / 1000: objective 173.162323
iteration 500 / 1000: objective 166.801300
iteration 600 / 1000: objective 161.737320
iteration 700 / 1000: objective 159.065781
```

```
iteration 800 / 1000: objective 158.405273
iteration 900 / 1000: objective 152.103226
iteration 0 / 1000: objective 229.329254
iteration 100 / 1000: objective 213.534485
iteration 200 / 1000: objective 201.638916
iteration 300 / 1000: objective 196.701996
iteration 400 / 1000: objective 192.575943
iteration 500 / 1000: objective 180.126709
iteration 600 / 1000: objective 171.558304
iteration 700 / 1000: objective 160.151138
iteration 800 / 1000: objective 163.994598
iteration 900 / 1000: objective 149.085739
Running weight scale=0.007848 at round 10 / 20
iteration 0 / 1000: objective 230.764557
iteration 100 / 1000: objective 210.515564
iteration 200 / 1000: objective 202.812622
iteration 300 / 1000: objective 193.124847
iteration 400 / 1000: objective 187.993668
iteration 500 / 1000: objective 175.584732
iteration 600 / 1000: objective 172.952744
iteration 700 / 1000: objective 166.806870
iteration 800 / 1000: objective 149.040985
iteration 900 / 1000: objective 149.694794
iteration 0 / 1000: objective 232.472290
iteration 100 / 1000: objective 209.607529
iteration 200 / 1000: objective 199.345398
iteration 300 / 1000: objective 199.686844
iteration 400 / 1000: objective 190.480118
iteration 500 / 1000: objective 175.075333
iteration 600 / 1000: objective 171.799835
iteration 700 / 1000: objective 160.107269
iteration 800 / 1000: objective 157.162048
iteration 900 / 1000: objective 143.635330
Running weight scale=0.012743 at round 11 / 20
iteration 0 / 1000: objective 232.220718
iteration 100 / 1000: objective 212.309021
iteration 200 / 1000: objective 195.058472
iteration 300 / 1000: objective 184.207657
iteration 400 / 1000: objective 187.166168
iteration 500 / 1000: objective 177.938171
iteration 600 / 1000: objective 174.394257
iteration 700 / 1000: objective 172.842499
iteration 800 / 1000: objective 159.449295
iteration 900 / 1000: objective 140.907425
iteration 0 / 1000: objective 231.457214
iteration 100 / 1000: objective 206.097366
iteration 200 / 1000: objective 200.768021
iteration 300 / 1000: objective 187.192017
```

```
iteration 400 / 1000: objective 184.854553
iteration 500 / 1000: objective 184.036774
iteration 600 / 1000: objective 172.598755
iteration 700 / 1000: objective 165.687866
iteration 800 / 1000: objective 158.084579
iteration 900 / 1000: objective 154.068054
Running weight scale=0.020691 at round 12 / 20
iteration 0 / 1000: objective 231.205994
iteration 100 / 1000: objective 212.729767
iteration 200 / 1000: objective 203.248734
iteration 300 / 1000: objective 189.998428
iteration 400 / 1000: objective 180.655243
iteration 500 / 1000: objective 177.685913
iteration 600 / 1000: objective 162.857620
iteration 700 / 1000: objective 154.503922
iteration 800 / 1000: objective 142.213913
iteration 900 / 1000: objective 133.419449
iteration 0 / 1000: objective 230.952682
iteration 100 / 1000: objective 217.639511
iteration 200 / 1000: objective 201.292526
iteration 300 / 1000: objective 198.220306
iteration 400 / 1000: objective 184.587280
iteration 500 / 1000: objective 176.618423
iteration 600 / 1000: objective 168.145981
iteration 700 / 1000: objective 159.556183
iteration 800 / 1000: objective 159.857498
iteration 900 / 1000: objective 146.640213
Running weight scale=0.033598 at round 13 / 20
iteration 0 / 1000: objective 230.868286
iteration 100 / 1000: objective 205.236389
iteration 200 / 1000: objective 204.174362
iteration 300 / 1000: objective 193.372620
iteration 400 / 1000: objective 183.155167
iteration 500 / 1000: objective 172.035843
iteration 600 / 1000: objective 167.069550
iteration 700 / 1000: objective 165.864532
iteration 800 / 1000: objective 156.998856
iteration 900 / 1000: objective 150.039032
iteration 0 / 1000: objective 229.857712
iteration 100 / 1000: objective 207.186691
iteration 200 / 1000: objective 201.593521
iteration 300 / 1000: objective 188.434280
iteration 400 / 1000: objective 179.571533
iteration 500 / 1000: objective 172.849991
iteration 600 / 1000: objective 168.775955
iteration 700 / 1000: objective 159.421341
iteration 800 / 1000: objective 152.954544
iteration 900 / 1000: objective 152.139969
```

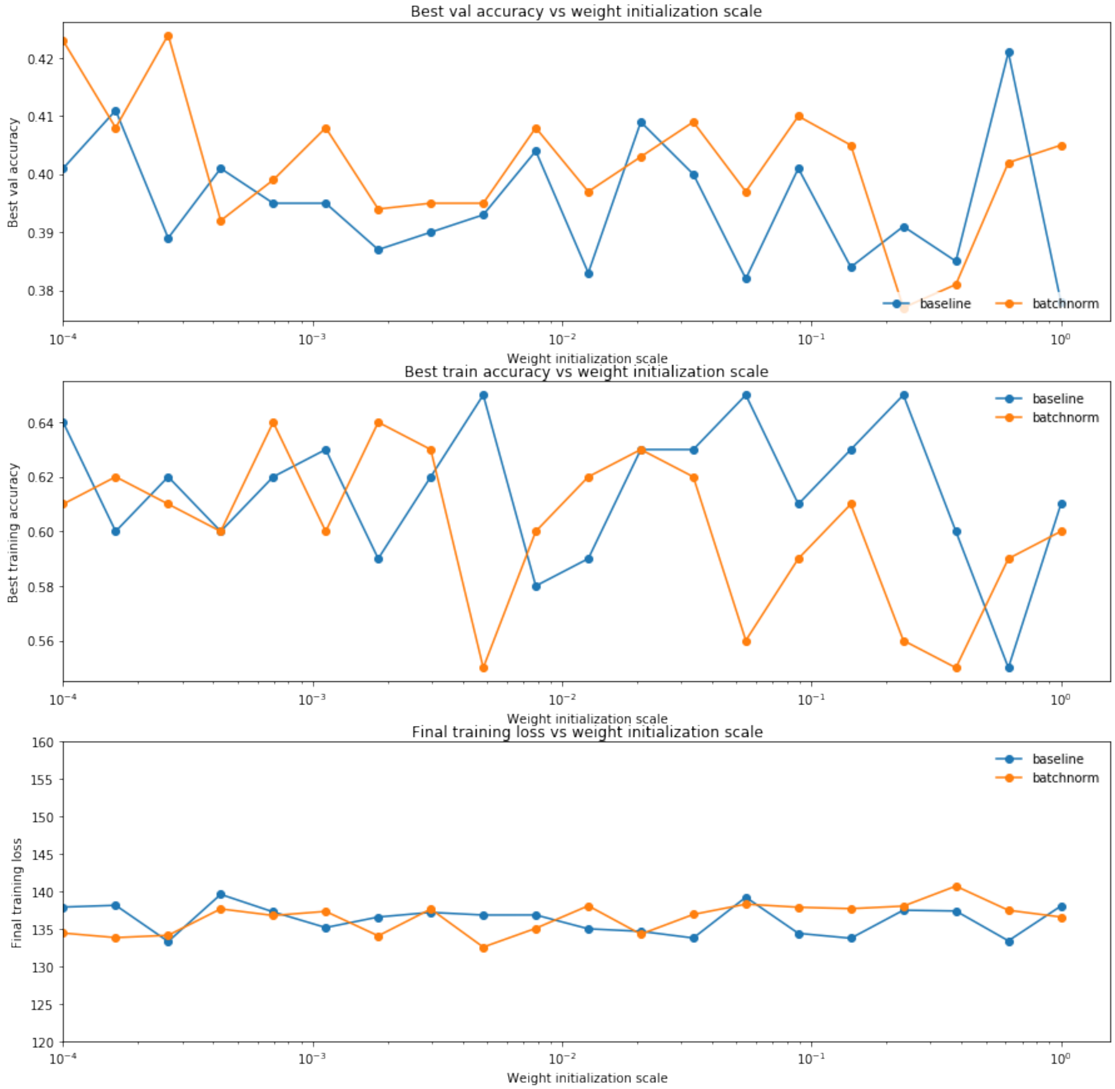
```
Running weight scale=0.054556 at round 14 / 20
iteration 0 / 1000: objective 230.686111
iteration 100 / 1000: objective 214.304245
iteration 200 / 1000: objective 211.966537
iteration 300 / 1000: objective 198.176468
iteration 400 / 1000: objective 185.245941
iteration 500 / 1000: objective 178.586349
iteration 600 / 1000: objective 172.485870
iteration 700 / 1000: objective 158.195816
iteration 800 / 1000: objective 153.962997
iteration 900 / 1000: objective 155.353409
iteration 0 / 1000: objective 231.731689
iteration 100 / 1000: objective 212.197540
iteration 200 / 1000: objective 205.097183
iteration 300 / 1000: objective 204.434875
iteration 400 / 1000: objective 188.473129
iteration 500 / 1000: objective 178.132904
iteration 600 / 1000: objective 171.941025
iteration 700 / 1000: objective 160.096512
iteration 800 / 1000: objective 153.947876
iteration 900 / 1000: objective 146.496857
Running weight scale=0.088587 at round 15 / 20
iteration 0 / 1000: objective 229.332108
iteration 100 / 1000: objective 211.046326
iteration 200 / 1000: objective 206.711334
iteration 300 / 1000: objective 186.463806
iteration 400 / 1000: objective 185.848755
iteration 500 / 1000: objective 175.506134
iteration 600 / 1000: objective 164.130844
iteration 700 / 1000: objective 155.263351
iteration 800 / 1000: objective 152.714905
iteration 900 / 1000: objective 146.926987
iteration 0 / 1000: objective 230.079987
iteration 100 / 1000: objective 205.132858
iteration 200 / 1000: objective 200.044418
iteration 300 / 1000: objective 191.371277
iteration 400 / 1000: objective 184.819946
iteration 500 / 1000: objective 176.852737
iteration 600 / 1000: objective 171.482956
iteration 700 / 1000: objective 162.837830
iteration 800 / 1000: objective 153.463318
iteration 900 / 1000: objective 149.009262
Running weight scale=0.143845 at round 16 / 20
iteration 0 / 1000: objective 230.316772
iteration 100 / 1000: objective 211.670013
iteration 200 / 1000: objective 200.105011
iteration 300 / 1000: objective 191.380630
iteration 400 / 1000: objective 181.944916
```

```
iteration 500 / 1000: objective 170.923584
iteration 600 / 1000: objective 170.367447
iteration 700 / 1000: objective 166.418396
iteration 800 / 1000: objective 161.339508
iteration 900 / 1000: objective 152.332870
iteration 0 / 1000: objective 229.165131
iteration 100 / 1000: objective 207.317566
iteration 200 / 1000: objective 194.481812
iteration 300 / 1000: objective 185.635376
iteration 400 / 1000: objective 181.975464
iteration 500 / 1000: objective 165.246338
iteration 600 / 1000: objective 153.887543
iteration 700 / 1000: objective 151.629135
iteration 800 / 1000: objective 144.158295
iteration 900 / 1000: objective 132.927917
Running weight scale=0.233572 at round 17 / 20
iteration 0 / 1000: objective 232.182251
iteration 100 / 1000: objective 212.789413
iteration 200 / 1000: objective 200.318634
iteration 300 / 1000: objective 202.714005
iteration 400 / 1000: objective 185.760773
iteration 500 / 1000: objective 177.377823
iteration 600 / 1000: objective 175.720245
iteration 700 / 1000: objective 161.823853
iteration 800 / 1000: objective 159.370285
iteration 900 / 1000: objective 150.915909
iteration 0 / 1000: objective 230.069275
iteration 100 / 1000: objective 205.765610
iteration 200 / 1000: objective 195.957474
iteration 300 / 1000: objective 186.526062
iteration 400 / 1000: objective 177.041214
iteration 500 / 1000: objective 169.967285
iteration 600 / 1000: objective 164.026398
iteration 700 / 1000: objective 167.641769
iteration 800 / 1000: objective 156.726868
iteration 900 / 1000: objective 157.090057
Running weight scale=0.379269 at round 18 / 20
iteration 0 / 1000: objective 231.338150
iteration 100 / 1000: objective 209.985062
iteration 200 / 1000: objective 207.838943
iteration 300 / 1000: objective 196.718582
iteration 400 / 1000: objective 192.502991
iteration 500 / 1000: objective 183.431686
iteration 600 / 1000: objective 179.082382
iteration 700 / 1000: objective 168.786804
iteration 800 / 1000: objective 157.824417
iteration 900 / 1000: objective 155.353149
iteration 0 / 1000: objective 229.827728
```

```
iteration 100 / 1000: objective 210.950729
iteration 200 / 1000: objective 209.711472
iteration 300 / 1000: objective 193.206161
iteration 400 / 1000: objective 186.360153
iteration 500 / 1000: objective 178.847321
iteration 600 / 1000: objective 165.292862
iteration 700 / 1000: objective 166.043015
iteration 800 / 1000: objective 154.336807
iteration 900 / 1000: objective 151.026367
Running weight scale=0.615848 at round 19 / 20
iteration 0 / 1000: objective 230.015930
iteration 100 / 1000: objective 214.884888
iteration 200 / 1000: objective 205.483612
iteration 300 / 1000: objective 195.105408
iteration 400 / 1000: objective 184.144958
iteration 500 / 1000: objective 177.880203
iteration 600 / 1000: objective 170.138092
iteration 700 / 1000: objective 171.366562
iteration 800 / 1000: objective 170.010498
iteration 900 / 1000: objective 162.500381
iteration 0 / 1000: objective 229.280289
iteration 100 / 1000: objective 212.673126
iteration 200 / 1000: objective 195.551605
iteration 300 / 1000: objective 193.985931
iteration 400 / 1000: objective 181.806381
iteration 500 / 1000: objective 174.012070
iteration 600 / 1000: objective 170.764938
iteration 700 / 1000: objective 163.804092
iteration 800 / 1000: objective 155.760193
iteration 900 / 1000: objective 155.719528
Running weight scale=1.000000 at round 20 / 20
iteration 0 / 1000: objective 229.734741
iteration 100 / 1000: objective 222.785873
iteration 200 / 1000: objective 200.033096
iteration 300 / 1000: objective 197.418213
iteration 400 / 1000: objective 189.376358
iteration 500 / 1000: objective 173.475937
iteration 600 / 1000: objective 165.583130
iteration 700 / 1000: objective 155.741241
iteration 800 / 1000: objective 157.053452
iteration 900 / 1000: objective 146.473969
iteration 0 / 1000: objective 230.125626
iteration 100 / 1000: objective 219.201172
iteration 200 / 1000: objective 204.317062
iteration 300 / 1000: objective 190.223694
iteration 400 / 1000: objective 184.143295
iteration 500 / 1000: objective 175.439423
iteration 600 / 1000: objective 169.510208
```

```
iteration 700 / 1000: objective 166.466385
iteration 800 / 1000: objective 163.659866
iteration 900 / 1000: objective 161.418411
```

In [46]:
```python
# 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(baseline_ws[ws]['train_acc_history']))
  bn_best_train_accs.append(max(bn_net_ws[ws]['train_acc_history']))

  best_val_accs.append(max(baseline_ws[ws]['val_acc_history']))
  bn_best_val_accs.append(max(bn_net_ws[ws]['val_acc_history']))

  final_train_loss.append(np.mean(baseline_ws[ws]['objective_history'][-100:]))
  bn_final_train_loss.append(np.mean(bn_net_ws[ws]['objective_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.gca().set_ylim(120, 160)

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



3.1 Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

3.2 Answer:

The scale of weight initialization should affect a model without batch normalization much more than one with batchnormalization, since the process of using batch normalization is in part meant to cancel out large variations in scale.

4 Batch normalization and batch size

We will now run a small experiment to study the interaction of batch normalization and batch size.

The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second cell will plot training accuracy and validation set accuracy over time.

Here is a link about batch sizes in batch normalization: https://www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks

```
In [50]: def run_batchsize_experiments():
             np.random.seed(15009)
             # Try training a very deep net with batchnorm
             hidden_dims = [50, 50, 50, 50, 50]

             num_train = 1000

             X_train = data['X_train'][:num_train]
             X_train = np.reshape(X_train, [X_train.shape[0], -1])
             y_train = data['y_train'][:num_train]

             X_val = data['X_val']
             X_val = np.reshape(X_val, [X_val.shape[0], -1])
             y_val = data['y_val']

             num_epochs = 10
             batch_sizes = [5,10,50]


             batch_size = batch_sizes[0]
             print('No normalization: batch size = ', 5)
             baseline = FullyConnectedNet(input_size=X_train.shape[1],
                                          hidden_size=hidden_dims,
                                          output_size=10,
                                          centering_data=True,
                                          use_dropout=False,
                                          use_bn=False)

             # use an aggresive learning rate
             baseline_trace = baseline.train(X_train, y_train, X_val, y_val,
                                             learning_rate=10**-3,
                                             reg=np.float32(1e-5),
                                             num_iters=num_train * num_epochs // batch_size ,
                                             batch_size=batch_size,
                                             verbose=True) # train the model with batch normal
```

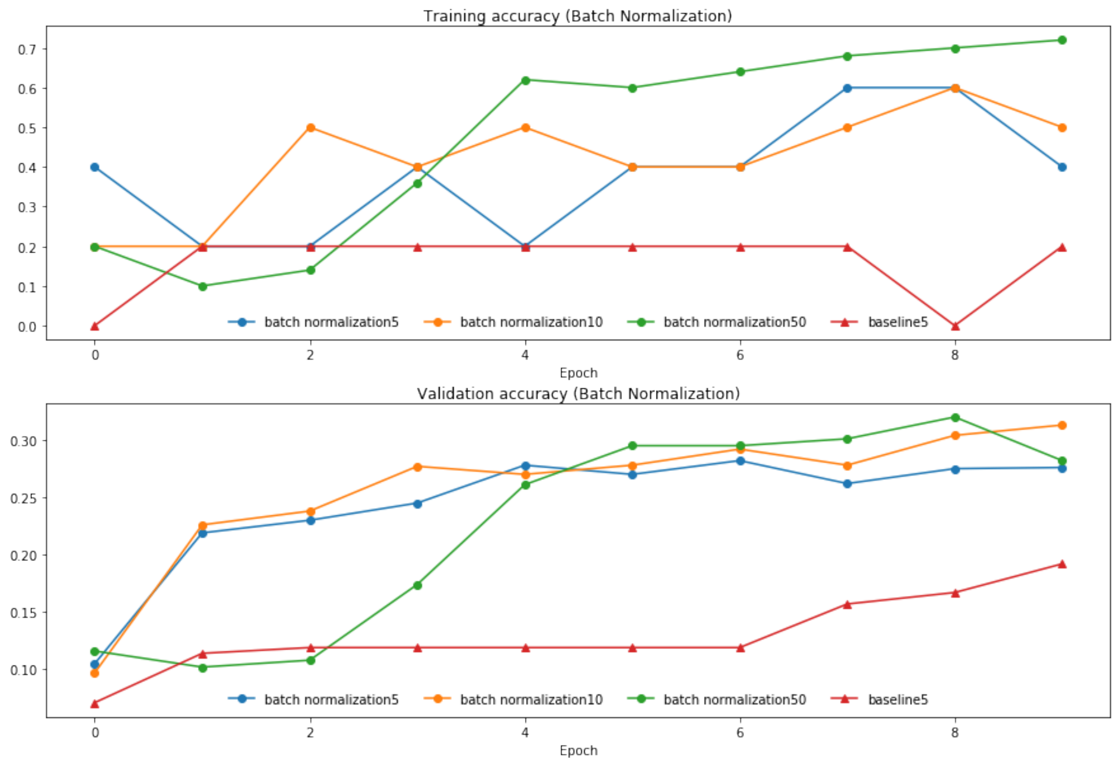
```
    bn_traces = []
    for i in range(len(batch_sizes)):

        batch_size = batch_sizes[i]
        print('Normalization: batch size = ',batch_size)


        bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                                     hidden_size=hidden_dims,
                                     output_size=10,
                                     centering_data=True,
                                     use_dropout=False,
                                     use_bn=True)

        # use an aggresive learning rate
        bn_net_trace = bn_model.train(X_train, y_train, X_val, y_val,
                                  learning_rate=10**-3,
                                  reg=np.float32(1e-5),
                                  num_iters=num_train * num_epochs // batch_size ,
                                  batch_size=batch_size,
                                  verbose=True) # train the model with batch normalization


        bn_traces.append(bn_net_trace)

    return bn_traces, baseline_trace, batch_sizes

batch_sizes = [5,10,50]
bn_traces, baseline_trace, batch_sizes = run_batchsize_experiments()
```

```
No normalization: batch size =  5
iteration 0 / 2000: objective 11.514305
iteration 100 / 2000: objective 11.530845
iteration 200 / 2000: objective 11.489232
iteration 300 / 2000: objective 11.541492
iteration 400 / 2000: objective 11.467241
iteration 500 / 2000: objective 11.549950
iteration 600 / 2000: objective 11.446713
iteration 700 / 2000: objective 11.555372
iteration 800 / 2000: objective 11.426662
iteration 900 / 2000: objective 11.555120
iteration 1000 / 2000: objective 11.404947
iteration 1100 / 2000: objective 11.544063
iteration 1200 / 2000: objective 11.375858
iteration 1300 / 2000: objective 11.504647
iteration 1400 / 2000: objective 11.316720
```

```
iteration 1500 / 2000: objective 11.314795
iteration 1600 / 2000: objective 11.028091
iteration 1700 / 2000: objective 9.682540
iteration 1800 / 2000: objective 10.528839
iteration 1900 / 2000: objective 9.623592
Normalization: batch size =  5
iteration 0 / 2000: objective 11.153890
iteration 100 / 2000: objective 11.320135
iteration 200 / 2000: objective 10.608068
iteration 300 / 2000: objective 10.766259
iteration 400 / 2000: objective 10.212790
iteration 500 / 2000: objective 10.212909
iteration 600 / 2000: objective 10.001297
iteration 700 / 2000: objective 9.883205
iteration 800 / 2000: objective 9.387010
iteration 900 / 2000: objective 9.481272
iteration 1000 / 2000: objective 9.506968
iteration 1100 / 2000: objective 10.199645
iteration 1200 / 2000: objective 8.730618
iteration 1300 / 2000: objective 9.562894
iteration 1400 / 2000: objective 9.570872
iteration 1500 / 2000: objective 10.009228
iteration 1600 / 2000: objective 8.468286
iteration 1700 / 2000: objective 9.959155
iteration 1800 / 2000: objective 7.707599
iteration 1900 / 2000: objective 8.987892
Normalization: batch size =  10
iteration 0 / 1000: objective 22.829689
iteration 100 / 1000: objective 22.076612
iteration 200 / 1000: objective 19.644665
iteration 300 / 1000: objective 18.531843
iteration 400 / 1000: objective 16.723150
iteration 500 / 1000: objective 14.642860
iteration 600 / 1000: objective 14.209229
iteration 700 / 1000: objective 12.041428
iteration 800 / 1000: objective 13.012772
iteration 900 / 1000: objective 11.496243
Normalization: batch size =  50
iteration 0 / 200: objective 115.275673
iteration 100 / 200: objective 80.218941
```

In [51]:
```python
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',
                      baseline_trace['train_acc_history'],
                      [trace['train_acc_history'] for trace in bn_traces],
                      bl_marker='-^', bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
```

```
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',
                      baseline_trace['val_acc_history'],
                      [trace['val_acc_history'] for trace in bn_traces],
                      bl_marker='-^', bn_marker='-o', labels=batch_sizes)

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



4.1 Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

4.2 Answer:

Large batch size seems to allow for better training accuracy but validation accuracy for smaller batch batch sizes with batch normalization seems to converge quicker and possibly to a higher value. This is probably because for smaller batch sizes we get less accurate estimates of the mean and variance from batch mean and variance, but smaller batch sizes also mean more training steps per epoch so we get faster convergence.