batch_normalization

March 2, 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)
```

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

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 updatec
        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:   [0.99999925]

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
[[0.61237198]]
```

```
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)
```

```
# 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_initializ
                                         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 normlaization 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

        print("Training batch %d: difference from the two implementations is %f" % (ibatch


    # 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, re
```

```
Training batch 0: difference from the two implementations is 0.000001
Training batch 1: difference from the two implementations is 0.000015
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 0.000000
Test batch 2: difference from the two implementations is 0.000002
```

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 [9]: 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 77.167534
iteration 200 / 800: objective 5.602989
iteration 300 / 800: objective 3.875721
iteration 400 / 800: objective 3.376601
iteration 500 / 800: objective 3.134349
iteration 600 / 800: objective 2.990509
iteration 700 / 800: objective 2.894545
```

Train a fully connected network without batch normalization

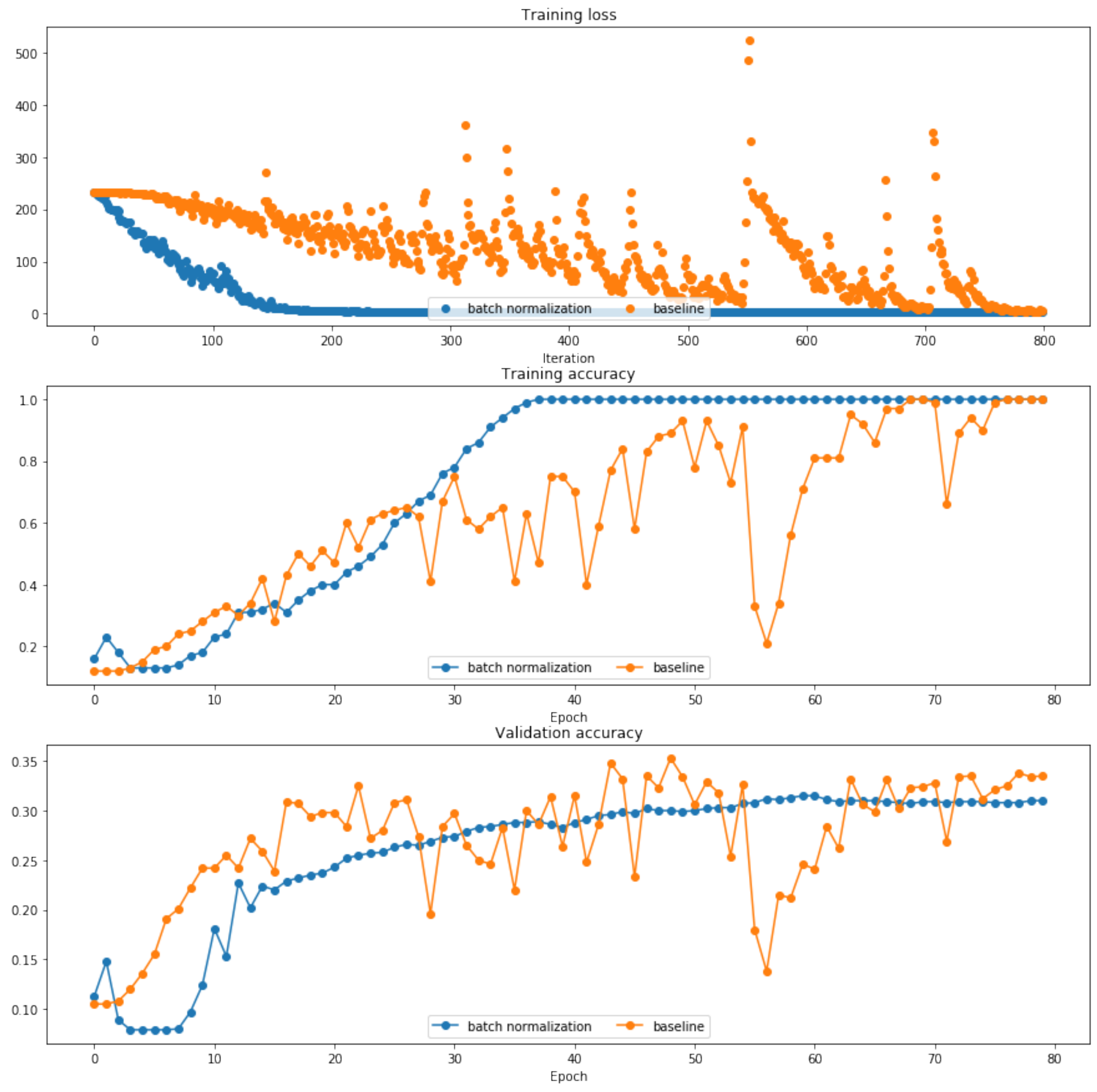
```
In [8]: model = FullyConnectedNet(input_size=X_train.shape[1],
                                  hidden_size=hidden_dims,
                                  output_size=10,
                                  centering_data=True,
                                  use_dropout=False,
                                  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 normalizatio
```

```
iteration 0 / 800: objective 232.689163
iteration 100 / 800: objective 202.023819
iteration 200 / 800: objective 153.214462
iteration 300 / 800: objective 119.032135
iteration 400 / 800: objective 82.500938
iteration 500 / 800: objective 76.225487
iteration 600 / 800: objective 83.255714
iteration 700 / 800: objective 12.773678
```

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 [10]: 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 [11]: 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
```

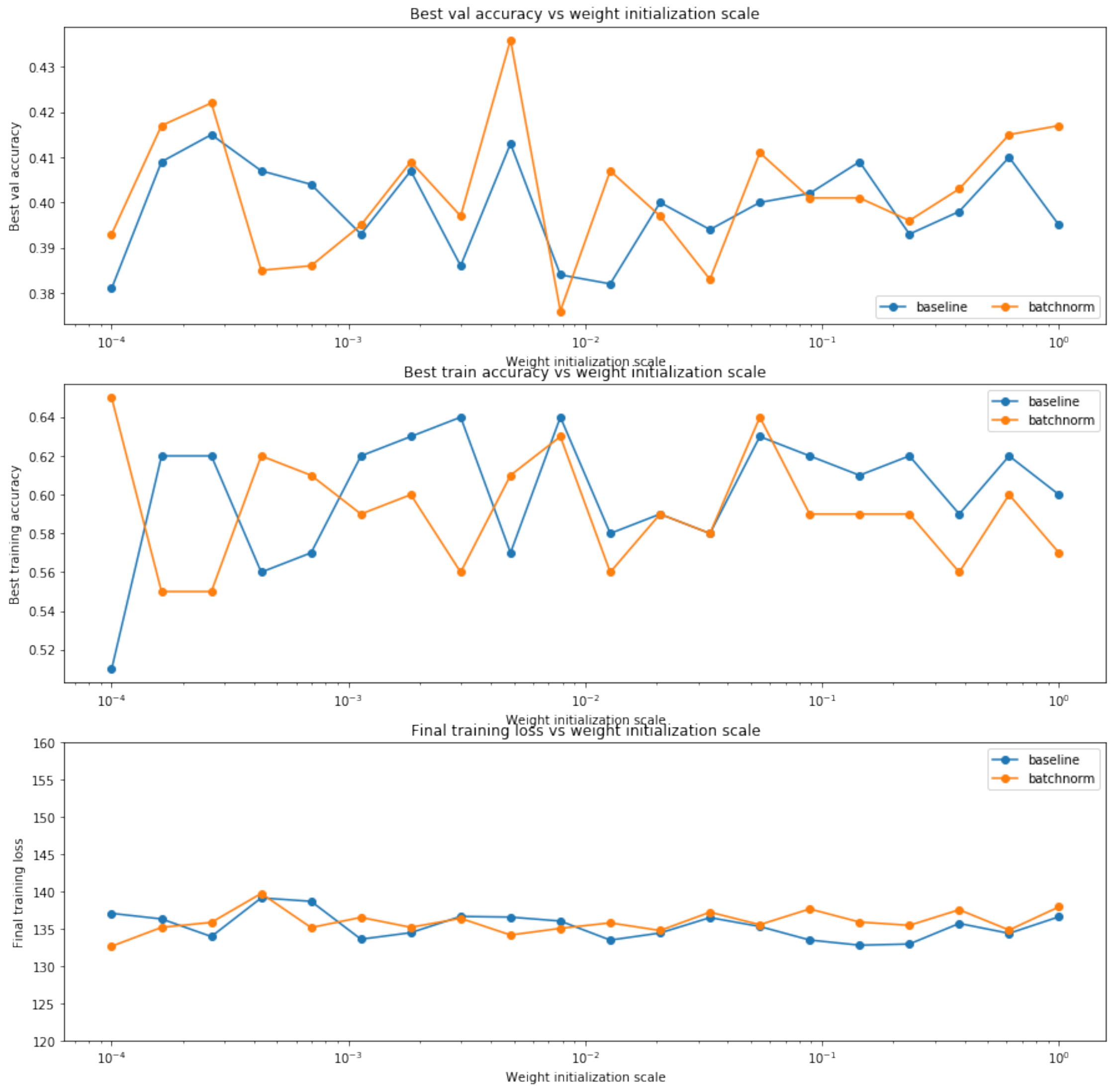
```
            model = FullyConnectedNet(input_size=X_train.shape[1],
                                      hidden_size=hidden_dims,
                                      output_size=10,
                                      centering_data=True,
                                      use_dropout=False,
                                      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 233.189835
iteration 100 / 1000: objective 209.775543
iteration 200 / 1000: objective 201.713638
iteration 300 / 1000: objective 184.172913
iteration 400 / 1000: objective 178.786423
iteration 500 / 1000: objective 174.454269
iteration 600 / 1000: objective 168.320297
iteration 700 / 1000: objective 161.511200
iteration 800 / 1000: objective 154.961670
iteration 900 / 1000: objective 151.681778
iteration 0 / 1000: objective 230.705185
iteration 100 / 1000: objective 212.256912
iteration 200 / 1000: objective 201.141724
iteration 300 / 1000: objective 189.029831
iteration 400 / 1000: objective 174.553085
iteration 500 / 1000: objective 172.012390
iteration 600 / 1000: objective 170.954239
iteration 700 / 1000: objective 170.689621
iteration 800 / 1000: objective 161.580795
iteration 900 / 1000: objective 155.987183
Running weight scale=0.000162 at round 2 / 20
iteration 0 / 1000: objective 231.216522
iteration 100 / 1000: objective 211.694092
iteration 200 / 1000: objective 197.277695
iteration 300 / 1000: objective 188.027969
iteration 400 / 1000: objective 182.634201
iteration 500 / 1000: objective 167.217438
iteration 600 / 1000: objective 157.979431
iteration 700 / 1000: objective 159.648193
iteration 800 / 1000: objective 155.507980
iteration 900 / 1000: objective 155.909622
iteration 0 / 1000: objective 234.763535
iteration 100 / 1000: objective 208.959473
iteration 200 / 1000: objective 203.447815
iteration 300 / 1000: objective 192.058060
iteration 400 / 1000: objective 182.993317
iteration 500 / 1000: objective 181.068008
iteration 600 / 1000: objective 174.770096
iteration 700 / 1000: objective 164.939743
iteration 800 / 1000: objective 154.621155
iteration 900 / 1000: objective 146.761475
Running weight scale=0.000264 at round 3 / 20
iteration 0 / 1000: objective 228.916122
iteration 100 / 1000: objective 210.116196
iteration 200 / 1000: objective 211.074356
iteration 300 / 1000: objective 197.934067
iteration 400 / 1000: objective 186.011475
iteration 500 / 1000: objective 180.971527
iteration 600 / 1000: objective 167.527573
iteration 700 / 1000: objective 164.871948
iteration 800 / 1000: objective 160.416000
iteration 900 / 1000: objective 142.544144
iteration 0 / 1000: objective 228.990707
iteration 100 / 1000: objective 209.663193
iteration 200 / 1000: objective 197.468094
iteration 300 / 1000: objective 185.401505
iteration 400 / 1000: objective 180.510345
iteration 500 / 1000: objective 174.146042
iteration 600 / 1000: objective 160.282379
iteration 700 / 1000: objective 144.068741
iteration 800 / 1000: objective 132.996628
iteration 900 / 1000: objective 134.049713
Running weight scale=0.000428 at round 4 / 20
iteration 0 / 1000: objective 230.535248
iteration 100 / 1000: objective 207.913208
iteration 200 / 1000: objective 197.349518
iteration 300 / 1000: objective 187.909180
iteration 400 / 1000: objective 185.955704
iteration 500 / 1000: objective 177.209457
iteration 600 / 1000: objective 171.816177
iteration 700 / 1000: objective 165.727234
iteration 800 / 1000: objective 154.905350
iteration 900 / 1000: objective 150.659424
iteration 0 / 1000: objective 230.348511
iteration 100 / 1000: objective 215.013306
iteration 200 / 1000: objective 206.962433
iteration 300 / 1000: objective 202.354050
iteration 400 / 1000: objective 191.837799
iteration 500 / 1000: objective 180.083939
iteration 600 / 1000: objective 175.603043
iteration 700 / 1000: objective 166.878891
iteration 800 / 1000: objective 166.461151
iteration 900 / 1000: objective 150.826813
Running weight scale=0.000695 at round 5 / 20
iteration 0 / 1000: objective 231.667969
iteration 100 / 1000: objective 217.241211
iteration 200 / 1000: objective 202.535431
iteration 300 / 1000: objective 191.557053
iteration 400 / 1000: objective 187.901657
iteration 500 / 1000: objective 175.559280
iteration 600 / 1000: objective 168.379852
iteration 700 / 1000: objective 168.055573
iteration 800 / 1000: objective 157.975967
iteration 900 / 1000: objective 147.270172
iteration 0 / 1000: objective 230.594971
iteration 100 / 1000: objective 229.583527
iteration 200 / 1000: objective 208.691406
iteration 300 / 1000: objective 200.446014
iteration 400 / 1000: objective 194.174805
iteration 500 / 1000: objective 181.498032
iteration 600 / 1000: objective 170.389923
iteration 700 / 1000: objective 163.235168
iteration 800 / 1000: objective 160.886536
iteration 900 / 1000: objective 161.293900
Running weight scale=0.001129 at round 6 / 20
iteration 0 / 1000: objective 231.056152
iteration 100 / 1000: objective 212.483292
iteration 200 / 1000: objective 201.244278
iteration 300 / 1000: objective 184.715546
iteration 400 / 1000: objective 179.207260
iteration 500 / 1000: objective 170.479874
iteration 600 / 1000: objective 164.907104
iteration 700 / 1000: objective 162.297241
iteration 800 / 1000: objective 154.656464
iteration 900 / 1000: objective 151.635834
iteration 0 / 1000: objective 230.562363
iteration 100 / 1000: objective 212.496353
iteration 200 / 1000: objective 198.897797
iteration 300 / 1000: objective 192.471695
iteration 400 / 1000: objective 186.708939
iteration 500 / 1000: objective 180.727310
iteration 600 / 1000: objective 162.368286
iteration 700 / 1000: objective 157.315323
iteration 800 / 1000: objective 156.595337
iteration 900 / 1000: objective 146.828049
Running weight scale=0.001833 at round 7 / 20
iteration 0 / 1000: objective 232.726456
iteration 100 / 1000: objective 214.330490
iteration 200 / 1000: objective 208.081146
iteration 300 / 1000: objective 192.803513
iteration 400 / 1000: objective 181.450836
iteration 500 / 1000: objective 174.167297
iteration 600 / 1000: objective 173.227585
iteration 700 / 1000: objective 159.641434
iteration 800 / 1000: objective 154.097321
iteration 900 / 1000: objective 145.930923
iteration 0 / 1000: objective 230.228195
iteration 100 / 1000: objective 222.838257
iteration 200 / 1000: objective 204.678146
iteration 300 / 1000: objective 194.680664
iteration 400 / 1000: objective 183.862930
iteration 500 / 1000: objective 174.341248
iteration 600 / 1000: objective 155.151535
iteration 700 / 1000: objective 144.407425
iteration 800 / 1000: objective 146.278320
iteration 900 / 1000: objective 140.492172
Running weight scale=0.002976 at round 8 / 20
iteration 0 / 1000: objective 231.871536
iteration 100 / 1000: objective 225.417923
iteration 200 / 1000: objective 207.787140
iteration 300 / 1000: objective 197.635117
iteration 400 / 1000: objective 194.951019
iteration 500 / 1000: objective 188.656021
iteration 600 / 1000: objective 172.265320
iteration 700 / 1000: objective 169.028412
iteration 800 / 1000: objective 159.704865
iteration 900 / 1000: objective 157.922607
iteration 0 / 1000: objective 230.618286
iteration 100 / 1000: objective 207.211090
iteration 200 / 1000: objective 198.678864
iteration 300 / 1000: objective 183.497528
iteration 400 / 1000: objective 176.186615
iteration 500 / 1000: objective 176.233368
iteration 600 / 1000: objective 166.789124
iteration 700 / 1000: objective 160.184204
iteration 800 / 1000: objective 156.439301
iteration 900 / 1000: objective 144.767136
Running weight scale=0.004833 at round 9 / 20
iteration 0 / 1000: objective 230.446899
iteration 100 / 1000: objective 208.288589
iteration 200 / 1000: objective 194.290070
iteration 300 / 1000: objective 175.748566
iteration 400 / 1000: objective 171.306442
iteration 500 / 1000: objective 169.819305
iteration 600 / 1000: objective 162.307495
iteration 700 / 1000: objective 155.560394
iteration 800 / 1000: objective 155.277267
iteration 900 / 1000: objective 145.316528
iteration 0 / 1000: objective 229.329254
iteration 100 / 1000: objective 216.092743
```

```
    ---------------------------------------------------------------------------

    KeyboardInterrupt                         Traceback (most recent call last)

    <ipython-input-11-672cd244ecd1> in <module>
     50                             num_iters=1000,
     51                             batch_size=100,
---> 52                             verbose=True)
     53


    ~/teaching/comp150dnn/assignments/comp150a2/implementations/fc_net.py in train(self, X
    302       np_objective, _, _ = session.run([self.operations['objective'],
    303                                         self.operations['bn_update'],
--> 304                                         self.operations['training_step']], feed_dict=
    305
    306       objective_history.append(np_objective)


    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in run(self, 
    927     try:
    928       result = self._run(None, fetches, feed_dict, options_ptr,
--> 929                          run_metadata_ptr)
    930       if run_metadata:
    931         proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)


    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in _run(self,
   1150     if final_fetches or final_targets or (handle and feed_dict_tensor):
   1151       results = self._do_run(handle, final_targets, final_fetches,
-> 1152                              feed_dict_tensor, options, run_metadata)
   1153     else:
   1154       results = []


    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in _do_run(se
   1326     if handle is None:
   1327       return self._do_call(_run_fn, feeds, fetches, targets, options,
-> 1328                            run_metadata)
   1329     else:
   1330       return self._do_call(_prun_fn, handle, feeds, fetches)
```

```
    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in _do_call(se
   1332   def _do_call(self, fn, *args):
   1333     try:
-> 1334       return fn(*args)
   1335     except errors.OpError as e:
   1336       message = compat.as_text(e.message)


    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in _run_fn(fe
   1317       self._extend_graph()
   1318       return self._call_tf_sessionrun(
-> 1319           options, feed_dict, fetch_list, target_list, run_metadata)
   1320
   1321     def _prun_fn(handle, feed_dict, fetch_list):


    ~/.local/lib/python3.5/site-packages/tensorflow/python/client/session.py in _call_tf_se
   1405     return tf_session.TF_SessionRun_wrapper(
   1406         self._session, options, feed_dict, fetch_list, target_list,
-> 1407         run_metadata)
   1408
   1409   def _call_tf_sessionprun(self, handle, feed_dict, fetch_list):


    KeyboardInterrupt:
```

```
In [166]: # 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:

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.

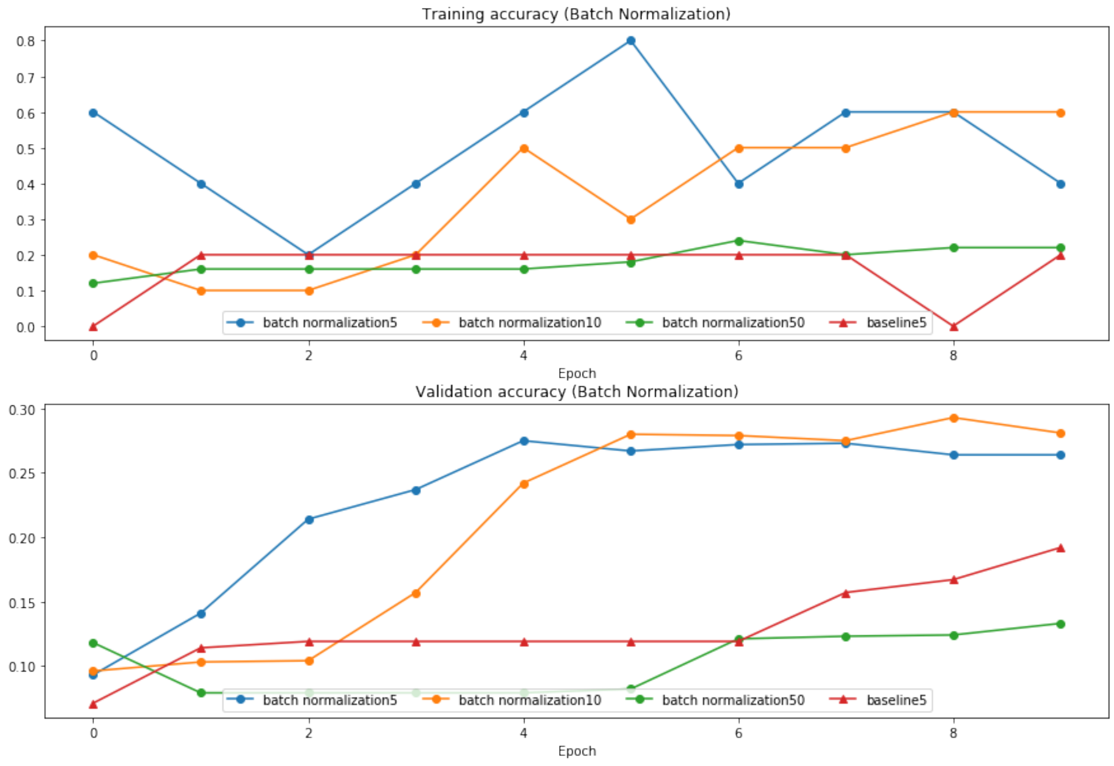
Here is a link about batch sizes in batch normalization: https://www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks

```
In [16]: 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 11.489233
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 11.426664
iteration 900 / 2000: objective 11.555120
iteration 1000 / 2000: objective 11.404947
iteration 1100 / 2000: objective 11.544064
iteration 1200 / 2000: objective 11.375858
iteration 1300 / 2000: objective 11.504647
iteration 1400 / 2000: objective 11.316722
iteration 1500 / 2000: objective 11.314794
iteration 1600 / 2000: objective 11.028092
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 11.153889
iteration 100 / 2000: objective 11.184366
iteration 200 / 2000: objective 10.620509
iteration 300 / 2000: objective 10.466232
iteration 400 / 2000: objective 9.730386
iteration 500 / 2000: objective 9.912413
iteration 600 / 2000: objective 9.729410
iteration 700 / 2000: objective 9.599819
iteration 800 / 2000: objective 9.428506
iteration 900 / 2000: objective 9.802075
iteration 1000 / 2000: objective 9.274168
iteration 1100 / 2000: objective 8.519082
iteration 1200 / 2000: objective 8.865942
iteration 1300 / 2000: objective 8.914768
iteration 1400 / 2000: objective 7.917655
iteration 1500 / 2000: objective 8.182160
iteration 1600 / 2000: objective 7.299488
iteration 1700 / 2000: objective 7.888865
iteration 1800 / 2000: objective 7.921083
iteration 1900 / 2000: objective 7.765578
Normalization: batch size =  10
iteration 0 / 1000: objective 22.829687
iteration 100 / 1000: objective 22.076609
iteration 200 / 1000: objective 19.644665
iteration 300 / 1000: objective 18.531832
iteration 400 / 1000: objective 16.723145
iteration 500 / 1000: objective 14.642863
iteration 600 / 1000: objective 14.604131
iteration 700 / 1000: objective 12.444086
iteration 800 / 1000: objective 12.967831
iteration 900 / 1000: objective 12.980310
Normalization: batch size =  50
iteration 0 / 200: objective 115.275681
iteration 100 / 200: objective 80.218948
```

```
In [17]: 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: