4 - BatchNormalization

March 21, 2019

1 Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization which was recently proposed by [3].

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [3] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [3] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

[3] 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
        from __future__ import print_function
        import time
        import numpy as np
        import matplotlib.pyplot as plt
        from libs.classifiers.fc_net import *
        from libs.data_utils import get_CIFAR10_data
        from libs.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
        from libs.solver import Solver

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

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

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

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

        data = get_CIFAR10_data()
        for k, v in data.items():
          print('%s: ' % k, v.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,)
```

1.1 Batch normalization: Forward

In the file `cs231n/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.

```
In [3]: # Check the training-time forward pass by checking means and variances
        # of features both before and after batch normalization

        # Simulate the forward pass for a two-layer network
        np.random.seed(231)
        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(a_norm)
        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:  [ -2.3814598  -13.18038246   1.91780462]
  stds:  [27.18502186 34.21455511 37.68611762]
After batch normalization (gamma=1, beta=0)
  mean:  [5.32907052e-17 7.04991621e-17 4.11476409e-17]
  std:  [0.99999999 1.         1.        ]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [0.99999999 1.99999999 2.99999999]
```

```
In [4]: # Check the test-time forward pass by running the training-time
        # forward pass many times to warm up the running averages, and then
        # checking the means and variances of activations after a test-time
        # forward pass.
        np.random.seed(231)
        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 range(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:  [-0.03927354 -0.04349152 -0.10452688]
  stds:  [1.01531428 1.01238373 0.97819988]
```

1.2 Batch Normalization: backward

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Once you have finished, run the following to numerically check your backward pass.

```
In [5]: # Gradient check batchnorm backward pass
        np.random.seed(231)
        N, D = 4, 5
        x = 5 * np.random.randn(N, D) + 12
        gamma = np.random.randn(D)
        beta = np.random.randn(D)
        dout = np.random.randn(N, D)

        bn_param = {'mode': 'train'}
        fx = lambda x: batchnorm_forward(x, gamma, beta, bn_param)[0]
        fg = lambda a: batchnorm_forward(x, a, beta, bn_param)[0]
        fb = lambda b: batchnorm_forward(x, gamma, b, bn_param)[0]

        dx_num = eval_numerical_gradient_array(fx, x, dout)
        da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
        db_num = eval_numerical_gradient_array(fb, beta.copy(), 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:  1.7029261167605239e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

1.3 Batch Normalization: alternative backward (OPTIONAL, +3 points extra credit)

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For the sigmoid function, it turns out that you can derive a very simple formula for the backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can also derive a simple expression for the batch normalization backward pass if you work out derivatives on paper and simplify. After doing so, implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

NOTE: This part of the assignment is entirely optional, but we will reward 3 points of extra credit if you can complete it.

```
In [6]: np.random.seed(231)
        N, D = 100, 500
        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'}
        out, cache = batchnorm_forward(x, gamma, beta, bn_param)

        t1 = time.time()
        dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
        t2 = time.time()
        dx2, dgamma2, dbeta2 = batchnorm_backward_alt(dout, cache)
        t3 = time.time()

        print('dx difference: ', rel_error(dx1, dx2))
        print('dgamma difference: ', rel_error(dgamma1, dgamma2))
        print('dbeta difference: ', rel_error(dbeta1, dbeta2))
        print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
```

```
dx difference:  9.352913533423052e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.57x
```

1.4 Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs2312n/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

Concretely, when the flag `use_batchnorm` is `True` in the constructor, you should insert a batch normalization layer before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized. Once you are done, run the following to gradient-check your implementation.

HINT: You might find it useful to define an additional helper layer similar to those in the file `cs231n/layer_utils.py`. If you decide to do so, do it in the file `cs231n/classifiers/fc_net.py`.

```
In [7]: np.random.seed(231)
        N, D, H1, H2, C = 2, 15, 20, 30, 10
        X = np.random.randn(N, D)
        y = np.random.randint(C, size=(N,))

        for reg in [0, 3.14]:
          print('Running check with reg = ', reg)
          model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                                    reg=reg, weight_scale=5e-2, dtype=np.float64,
                                    use_batchnorm=True)

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

          for name in sorted(grads):
            f = lambda _: model.loss(X, y)[0]
            grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
            print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
          if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 3.07e-06
W3 relative error: 3.92e-10
b1 relative error: 4.44e-08
b2 relative error: 2.22e-08
b3 relative error: 9.06e-11
beta1 relative error: 7.85e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 3.35e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.11e-08
b1 relative error: 5.55e-09
b2 relative error: 5.55e-09
b3 relative error: 1.73e-10
beta1 relative error: 6.65e-09
beta2 relative error: 3.48e-09
```

```
gamma1 relative error: 5.94e-09
gamma2 relative error: 4.67e-09
```

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 [13]: np.random.seed(231)
         # Try training a very deep net with batchnorm
         hidden_dims = [100, 100, 100, 100, 100]

         num_train = 1000
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': data['y_val'],
         }

         weight_scale = 2e-2
         bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=Tru
         model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=False

         bn_solver = Solver(bn_model, small_data,
                         num_epochs=10, batch_size=50,
                         update_rule='adam',
                         optim_config={
                           'learning_rate': 1e-3,
                         },
                         verbose=True, print_every=10)
         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=10)
         solver.train()
```

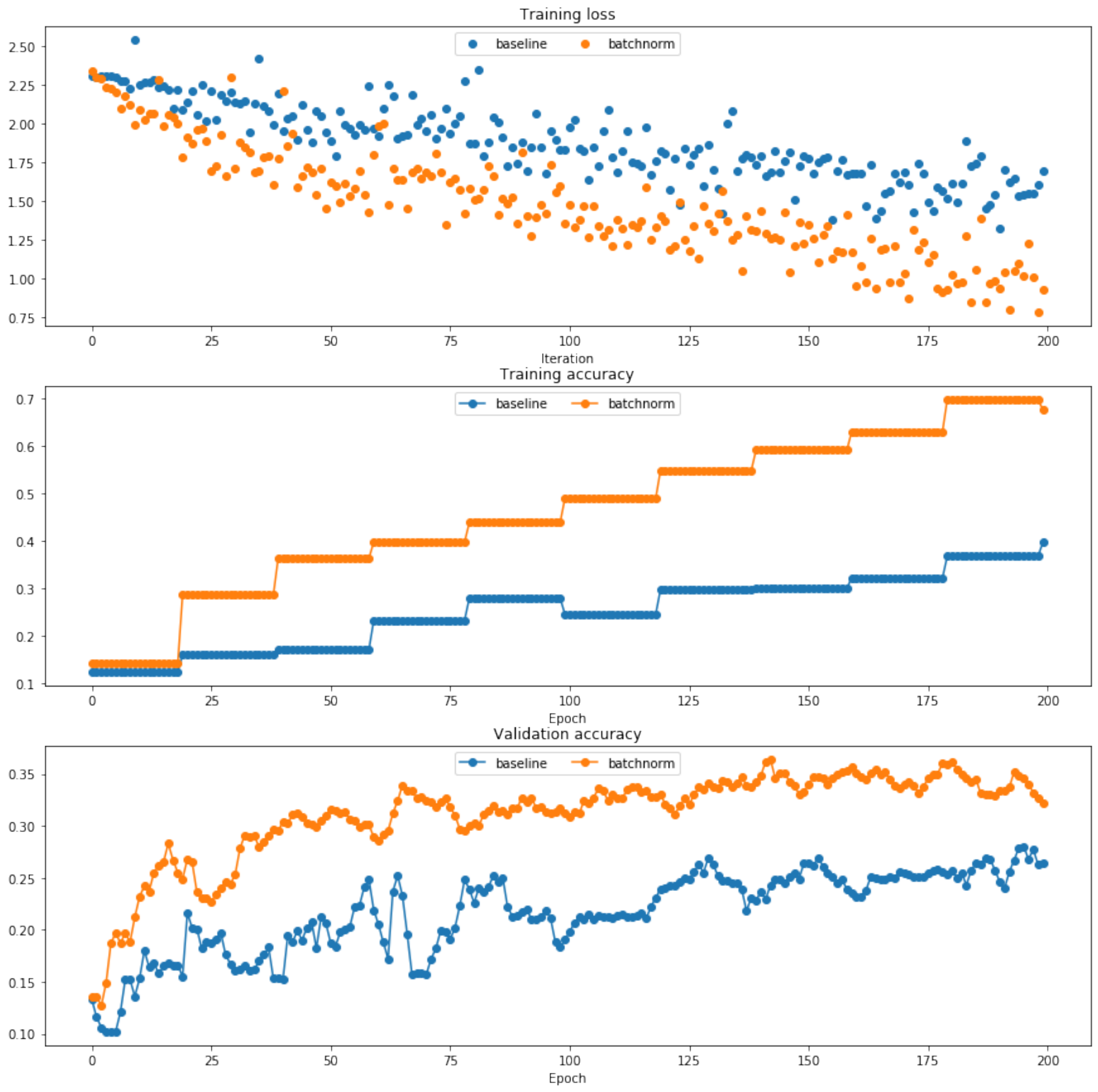
```
(Epoch 0 / 10) (Iteration 1 / 200) loss: 2.340975 train acc: 0.141000 val_acc: 0.135000
(Epoch 0 / 10) (Iteration 11 / 200) loss: 2.090345 train acc: 0.141000 val_acc: 0.232000
(Epoch 1 / 10) (Iteration 21 / 200) loss: 1.911842 train acc: 0.286000 val_acc: 0.268000
(Epoch 1 / 10) (Iteration 31 / 200) loss: 1.709355 train acc: 0.286000 val_acc: 0.253000
(Epoch 2 / 10) (Iteration 41 / 200) loss: 2.212336 train acc: 0.363000 val_acc: 0.304000
```

```
(Epoch 2 / 10) (Iteration 51 / 200) loss: 1.620877 train acc: 0.363000 val_acc: 0.316000
(Epoch 3 / 10) (Iteration 61 / 200) loss: 1.982238 train acc: 0.397000 val_acc: 0.286000
(Epoch 3 / 10) (Iteration 71 / 200) loss: 1.681823 train acc: 0.397000 val_acc: 0.324000
(Epoch 4 / 10) (Iteration 81 / 200) loss: 1.509997 train acc: 0.438000 val_acc: 0.302000
(Epoch 4 / 10) (Iteration 91 / 200) loss: 1.812851 train acc: 0.438000 val_acc: 0.327000
(Epoch 5 / 10) (Iteration 101 / 200) loss: 1.479101 train acc: 0.489000 val_acc: 0.308000
(Epoch 5 / 10) (Iteration 111 / 200) loss: 1.380561 train acc: 0.489000 val_acc: 0.326000
(Epoch 6 / 10) (Iteration 121 / 200) loss: 1.370221 train acc: 0.548000 val_acc: 0.321000
(Epoch 6 / 10) (Iteration 131 / 200) loss: 1.308173 train acc: 0.548000 val_acc: 0.337000
(Epoch 7 / 10) (Iteration 141 / 200) loss: 1.435489 train acc: 0.591000 val_acc: 0.348000
(Epoch 7 / 10) (Iteration 151 / 200) loss: 1.349083 train acc: 0.591000 val_acc: 0.340000
(Epoch 8 / 10) (Iteration 161 / 200) loss: 0.954339 train acc: 0.629000 val_acc: 0.351000
(Epoch 8 / 10) (Iteration 171 / 200) loss: 1.030413 train acc: 0.629000 val_acc: 0.340000
(Epoch 9 / 10) (Iteration 181 / 200) loss: 1.024786 train acc: 0.697000 val_acc: 0.362000
(Epoch 9 / 10) (Iteration 191 / 200) loss: 0.939608 train acc: 0.697000 val_acc: 0.334000
(Epoch 0 / 10) (Iteration 1 / 200) loss: 2.302332 train acc: 0.123000 val_acc: 0.133000
(Epoch 0 / 10) (Iteration 11 / 200) loss: 2.248739 train acc: 0.123000 val_acc: 0.153000
(Epoch 1 / 10) (Iteration 21 / 200) loss: 2.132461 train acc: 0.160000 val_acc: 0.216000
(Epoch 1 / 10) (Iteration 31 / 200) loss: 2.140029 train acc: 0.160000 val_acc: 0.161000
(Epoch 2 / 10) (Iteration 41 / 200) loss: 1.951723 train acc: 0.171000 val_acc: 0.152000
(Epoch 2 / 10) (Iteration 51 / 200) loss: 1.886314 train acc: 0.171000 val_acc: 0.187000
(Epoch 3 / 10) (Iteration 61 / 200) loss: 1.921923 train acc: 0.232000 val_acc: 0.205000
(Epoch 3 / 10) (Iteration 71 / 200) loss: 1.947953 train acc: 0.232000 val_acc: 0.157000
(Epoch 4 / 10) (Iteration 81 / 200) loss: 1.869925 train acc: 0.279000 val_acc: 0.226000
(Epoch 4 / 10) (Iteration 91 / 200) loss: 1.880315 train acc: 0.279000 val_acc: 0.217000
(Epoch 5 / 10) (Iteration 101 / 200) loss: 1.978349 train acc: 0.245000 val_acc: 0.198000
(Epoch 5 / 10) (Iteration 111 / 200) loss: 1.684166 train acc: 0.245000 val_acc: 0.214000
(Epoch 6 / 10) (Iteration 121 / 200) loss: 1.808940 train acc: 0.297000 val_acc: 0.240000
(Epoch 6 / 10) (Iteration 131 / 200) loss: 1.697445 train acc: 0.297000 val_acc: 0.263000
(Epoch 7 / 10) (Iteration 141 / 200) loss: 1.790114 train acc: 0.300000 val_acc: 0.237000
(Epoch 7 / 10) (Iteration 151 / 200) loss: 1.777415 train acc: 0.300000 val_acc: 0.264000
(Epoch 8 / 10) (Iteration 161 / 200) loss: 1.676356 train acc: 0.321000 val_acc: 0.232000
(Epoch 8 / 10) (Iteration 171 / 200) loss: 1.685098 train acc: 0.321000 val_acc: 0.255000
(Epoch 9 / 10) (Iteration 181 / 200) loss: 1.608762 train acc: 0.367000 val_acc: 0.257000
(Epoch 9 / 10) (Iteration 191 / 200) loss: 1.324150 train acc: 0.367000 val_acc: 0.246000
```

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 [14]: plt.subplot(3, 1, 1)
         plt.title('Training loss')
         plt.xlabel('Iteration')

         plt.subplot(3, 1, 2)
         plt.title('Training accuracy')
         plt.xlabel('Epoch')
         plt.subplot(3, 1, 3)
         plt.title('Validation accuracy')
         plt.xlabel('Epoch')

         plt.subplot(3, 1, 1)
         plt.plot(solver.loss_history, 'o', label='baseline')
         plt.plot(bn_solver.loss_history, 'o', label='batchnorm')

         plt.subplot(3, 1, 2)
         plt.plot(solver.train_acc_history, '-o', label='baseline')
         plt.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

         plt.subplot(3, 1, 3)
         plt.plot(solver.val_acc_history, '-o', label='baseline')
         plt.plot(bn_solver.val_acc_history, '-o', label='batchnorm')

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

```
/Users/lixingxuan/anaconda3/envs/tesnorflow/lib/python3.6/site-packages/matplotlib/cbook/depre
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



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 layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

```
In [17]: np.random.seed(231)
         # Try training a very deep net with batchnorm
         hidden_dims = [50, 50, 50, 50, 50, 50, 50]

         num_train = 1000
```

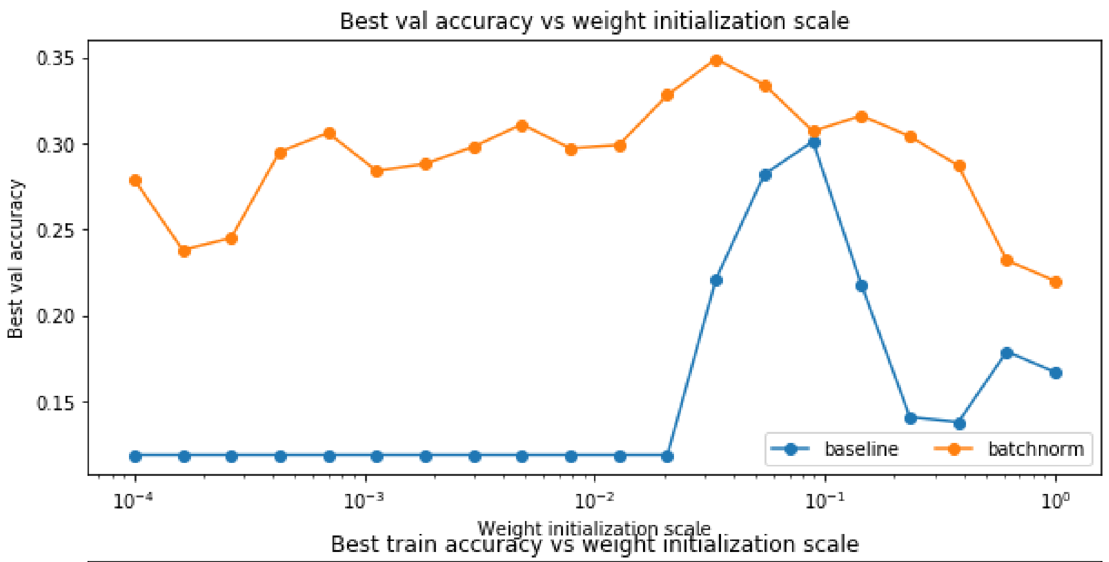
```
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

bn_solvers = {}
solvers = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
  bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=Fal

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

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

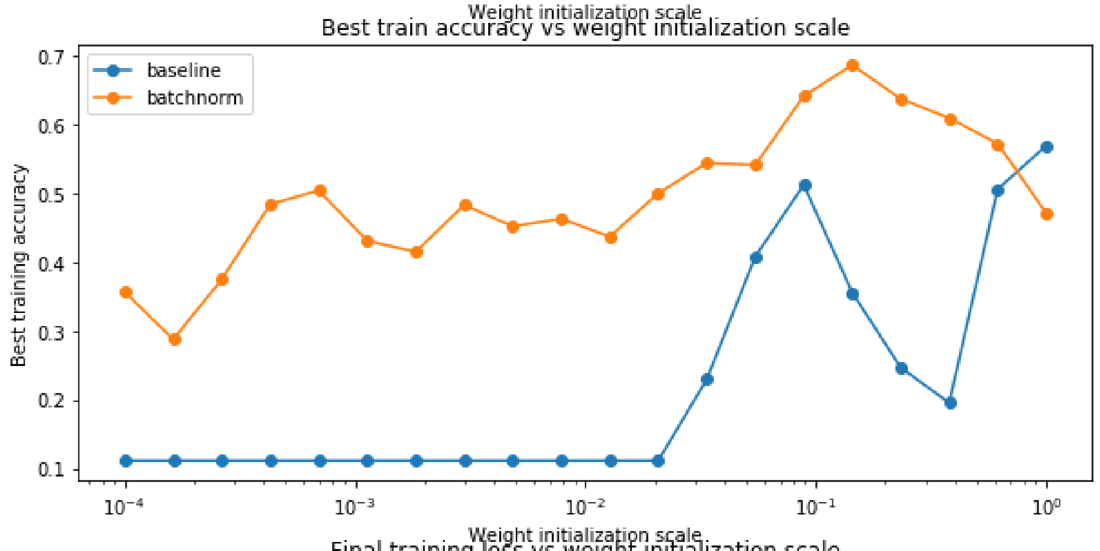
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale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

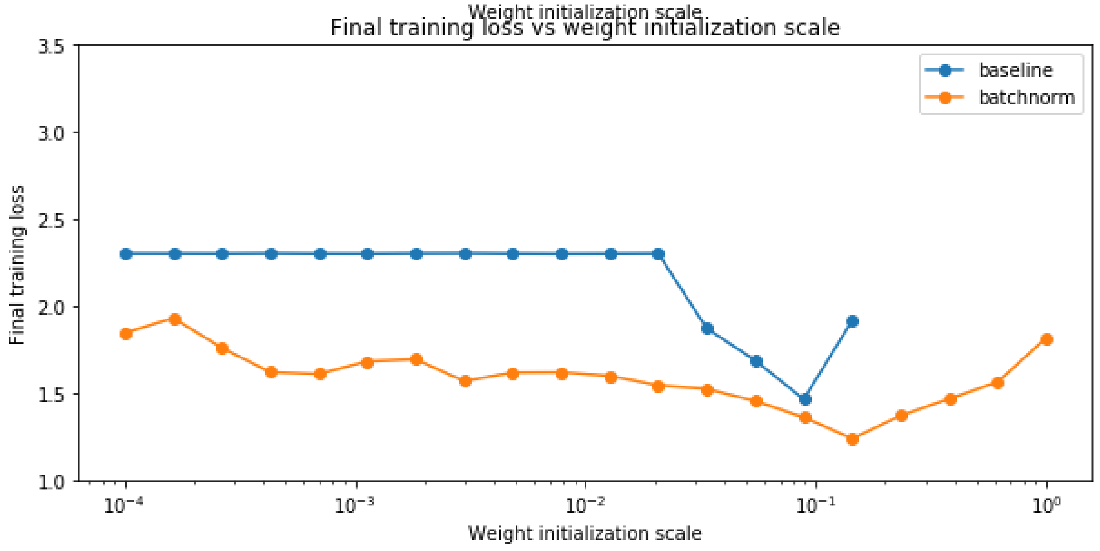
```
/Users/lixingxuan/Desktop/Computer Vision/lab2-forStudents/assignment2/libs/layers.py:1363: Ru
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
```

```
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

```
In [18]: # 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.gca().set_ylim(1.0, 3.5)

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


Best val accuracy vs weight initialization scale
Best val accuracy
0.35
0.30
0.25
0.20
0.15
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale


Best train accuracy vs weight initialization scale
Best training accuracy
0.7
0.6
0.5
0.4
0.3
0.2
0.1
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale


Final training loss vs weight initialization scale
Final training loss
3.5
3.0
2.5
2.0
1.5
1.0
baseline
batchnorm
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Weight initialization scale

4 Question:

Describe the results of this experiment, and try to give a reason why the experiment gave the results that it did.

5 Answer:

For both training and validation accuracy, the model with batchnorm has better performance than model without batch normalization. For the third graph, the model with batch normalization has less final training loss than the one without normalization. This is because that batch normalization makes the training more robust. It normalizes the magnitude of input values.