BatchNormalization

November 20, 2019

b. Q2(:Q2: Batch Normalization (30 points)) 의 결과를 작성한 코드와 함께 출력하세요. (10점)

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 proposed by [1] in 2015.

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 [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [1] 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.

[1] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

```
In [1]: # As usual, a bit of setup
        import time
        import numpy as np
        import matplotlib.pyplot as plt
        from cs231n.classifiers.fc_net import *
        from cs231n.data_utils import get_CIFAR10_data
```

```
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arra
from cs231n.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))))

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_train:  (49000,)
X_train:  (49000, 3, 32, 32)
X_val:  (1000, 3, 32, 32)
y_test:  (1000,)
X_test:  (1000, 3, 32, 32)
y_val:  (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.

Referencing the paper linked to above in [1] may be helpful!

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_mean_std(a,axis=0)

gamma = np.ones((D3,))
beta = np.zeros((D3,))
# Means should be close to zero and stds close to one
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
# Now means should be close to beta and stds close to gamma
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,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)
  means:  [5.32907052e-17 7.04991621e-17 1.85962357e-17]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  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_mean_std(a_norm,axis=0)
```

```
After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:   [1.01531427 1.01238373 0.97819987]
```

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)
```

```python
#You should expect to see relative errors between 1e-13 and 1e-8
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:  1.7029258328157158e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

1.3 Batch normalization: alternative backward

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 example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can do a similar simplification for the batch normalization backward pass too!

In the forward pass, given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}$,

we first calculate the mean μ and variance v. With μ and v calculated, we can calculate the standard deviation σ and normalized data Y. The equations and graph illustration below describe the computation (y_i is the i-th element of the vector Y).

$$\mu = \frac{1}{N}\sum_{k=1}^{N} x_k \qquad v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2 \tag{1}$$

$$\sigma = \sqrt{v + \epsilon} \qquad y_i = \frac{x_i - \mu}{\sigma} \tag{2}$$

The meat of our problem during backpropagation is to compute $\frac{\partial L}{\partial X}$, given the upstream gradient we receive, $\frac{\partial L}{\partial Y}$. To do this, recall the chain rule in calculus gives us $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X}$.

The unknown/hart part is $\frac{\partial Y}{\partial X}$. We can find this by first deriving step-by-step our local gradients at $\frac{\partial v}{\partial X}$, $\frac{\partial \mu}{\partial X}$, $\frac{\partial \sigma}{\partial v}$, $\frac{\partial Y}{\partial \sigma}$, and $\frac{\partial Y}{\partial \mu}$, and then use the chain rule to compose these gradients (which appear in the form of vectors!) appropriately to compute $\frac{\partial Y}{\partial X}$.

If it's challenging to directly reason about the gradients over X and Y which require matrix multiplication, try reasoning about the gradients in terms of individual elements x_i and y_i first: in that case, you will need to come up with the derivations for $\frac{\partial L}{\partial x_i}$, by relying on the Chain Rule to first calculate the intermediate $\frac{\partial \mu}{\partial x_i}, \frac{\partial v}{\partial x_i}, \frac{\partial \sigma}{\partial x_i}$, then assemble these pieces to calculate $\frac{\partial y_i}{\partial x_i}$.

You should make sure each of the intermediary gradient derivations are all as simplified as possible, for ease of implementation.

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.

```
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:  6.284600172572596e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.71x
```

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 `cs231n/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

Concretely, when the `normalization` flag is set to `"batchnorm"` 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,))

        # You should expect losses between 1e-4~1e-10 for W,
        # losses between 1e-08~1e-10 for b,
        # and losses between 1e-08~1e-09 for beta and gammas.
        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,
                          normalization='batchnorm')

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: 2.85e-06
W3 relative error: 4.05e-10
b1 relative error: 2.22e-07
b2 relative error: 2.22e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.33e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 6.96e-09
gamma2 relative error: 1.96e-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: 1.38e-08
b2 relative error: 7.99e-07
b3 relative error: 1.73e-10
beta1 relative error: 6.65e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 8.80e-09
gamma2 relative error: 5.28e-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 [8]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
```

```python
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, normalization='ba
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=None)

print('Solver with batch norm:')
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=20)
bn_solver.train()

print('\nSolver without batch norm:')
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

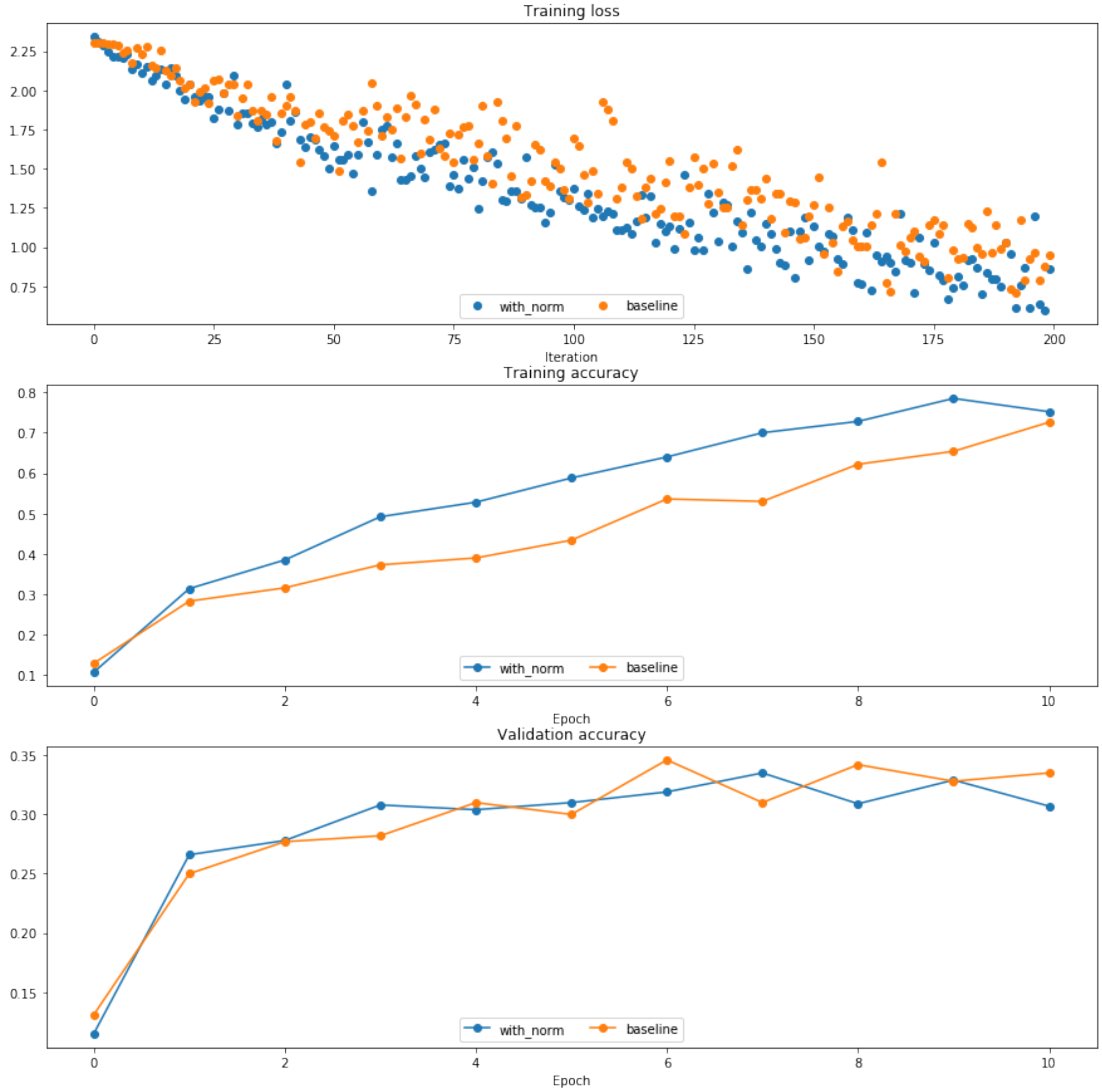
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340975
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.314000; val_acc: 0.266000
(Iteration 21 / 200) loss: 2.039365
(Epoch 2 / 10) train acc: 0.385000; val_acc: 0.278000
(Iteration 41 / 200) loss: 2.041103
(Epoch 3 / 10) train acc: 0.492000; val_acc: 0.308000
(Iteration 61 / 200) loss: 1.753902
(Epoch 4 / 10) train acc: 0.528000; val_acc: 0.304000
(Iteration 81 / 200) loss: 1.241217
(Epoch 5 / 10) train acc: 0.588000; val_acc: 0.310000
(Iteration 101 / 200) loss: 1.371013
(Epoch 6 / 10) train acc: 0.640000; val_acc: 0.319000
```

```
(Iteration 121 / 200) loss: 1.135679
(Epoch 7 / 10) train acc: 0.700000; val_acc: 0.335000
(Iteration 141 / 200) loss: 1.152127
(Epoch 8 / 10) train acc: 0.728000; val_acc: 0.309000
(Iteration 161 / 200) loss: 0.767138
(Epoch 9 / 10) train acc: 0.785000; val_acc: 0.329000
(Iteration 181 / 200) loss: 0.813475
(Epoch 10 / 10) train acc: 0.752000; val_acc: 0.307000

Solver without batch norm:
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.131000
(Epoch 1 / 10) train acc: 0.283000; val_acc: 0.250000
(Iteration 21 / 200) loss: 2.041970
(Epoch 2 / 10) train acc: 0.316000; val_acc: 0.277000
(Iteration 41 / 200) loss: 1.900473
(Epoch 3 / 10) train acc: 0.373000; val_acc: 0.282000
(Iteration 61 / 200) loss: 1.713156
(Epoch 4 / 10) train acc: 0.390000; val_acc: 0.310000
(Iteration 81 / 200) loss: 1.662209
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696062
(Epoch 6 / 10) train acc: 0.536000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.550785
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.310000
(Iteration 141 / 200) loss: 1.436308
(Epoch 8 / 10) train acc: 0.622000; val_acc: 0.342000
(Iteration 161 / 200) loss: 1.000868
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.328000
(Iteration 181 / 200) loss: 0.925455
(Epoch 10 / 10) train acc: 0.726000; val_acc: 0.335000
```

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 [9]: def plot_training_history(title, label, baseline, bn_solvers, plot_fn, bl_marker='.', b
            """utility function for plotting training history"""
            plt.title(title)
            plt.xlabel(label)
            bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
            bl_plot = plot_fn(baseline)
            num_bn = len(bn_plots)
            for i in range(num_bn):
                label='with_norm'
                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', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.train_acc_history, bl_marker='-o', bn_marker='-o')
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.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 layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

```
In [10]: 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 = {
```

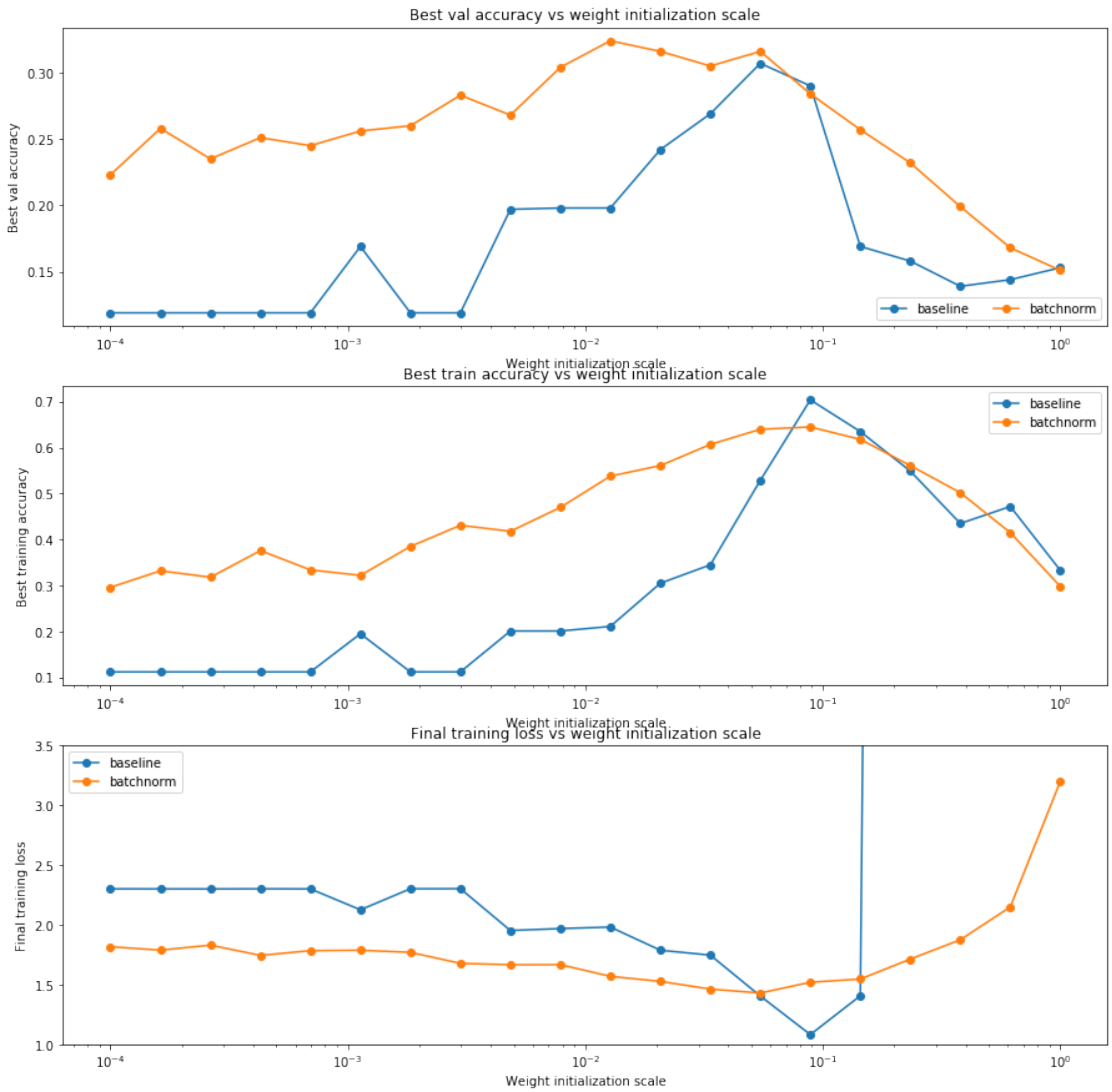
```
      'X_train': data['X_train'][:num_train],
      'y_train': data['y_train'][:num_train],
      'X_val': data['X_val'],
      'y_val': data['y_val'],
    }

    bn_solvers_ws = {}
    solvers_ws = {}
    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, normalization=
      model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=Non

      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_ws[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_ws[weight_scale] = solver
```

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

```
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

In [11]:
```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(solvers_ws[ws].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers_ws[ws].train_acc_history))

  best_val_accs.append(max(solvers_ws[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers_ws[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers_ws[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(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:

weight initialization에 모델의 성능은 batch normalization을 함께하는 경우 안하는 경우보다 덜 민감하다 위의 결과에서 보듯 batch normalization은 regularization의 효과를 보여주고 또한 exploding gradient의 문제에도 강건함을 보여준다.

4 Batch normalization and batch size

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

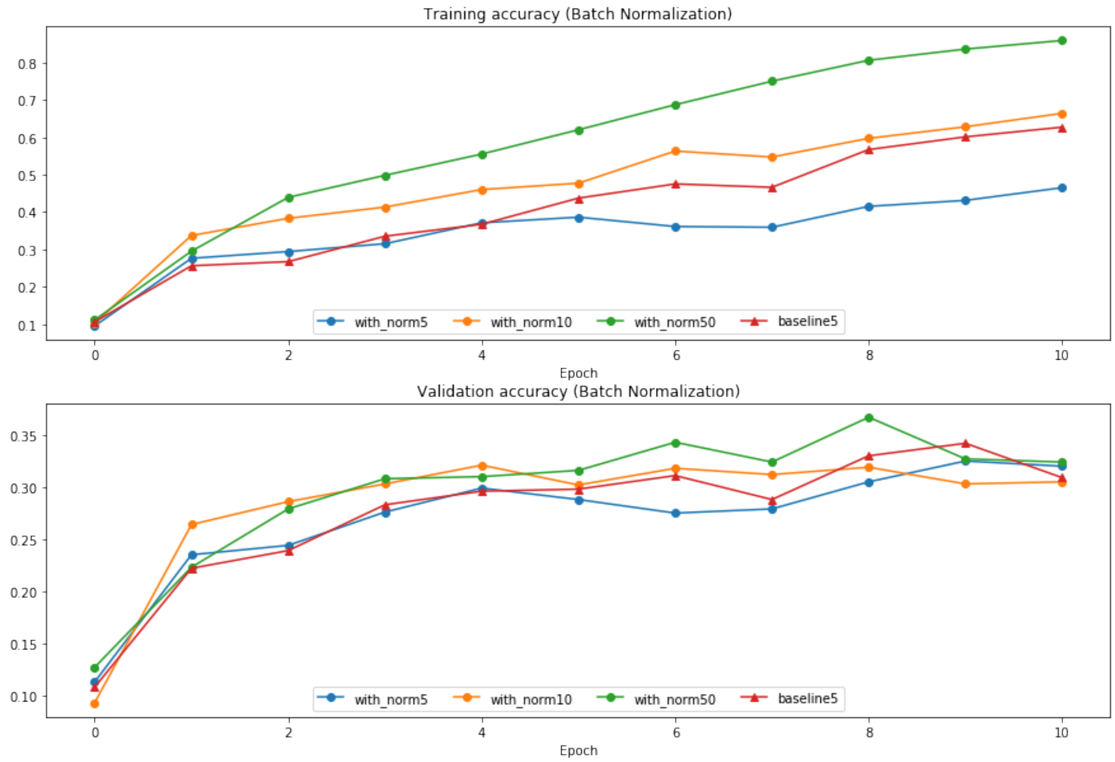
The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

```
In [12]: def run_batchsize_experiments(normalization_mode):
             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'],
             }
             n_epochs=10
             weight_scale = 2e-2
             batch_sizes = [5,10,50]
             lr = 10**(-3.5)
             solver_bsize = batch_sizes[0]

             print('No normalization: batch size = ',solver_bsize)
             model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=No
             solver = Solver(model, small_data,
                             num_epochs=n_epochs, batch_size=solver_bsize,
                             update_rule='adam',
                             optim_config={
                               'learning_rate': lr,
                             },
                             verbose=False)
             solver.train()

             bn_solvers = []
             for i in range(len(batch_sizes)):
                 b_size=batch_sizes[i]
                 print('Normalization: batch size = ',b_size)
                 bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normaliza
                 bn_solver = Solver(bn_model, small_data,
                                 num_epochs=n_epochs, batch_size=b_size,
                                 update_rule='adam',
                                 optim_config={
                                   'learning_rate': lr,
                                 },
                                 verbose=False)
                 bn_solver.train()
```

```
            bn_solvers.append(bn_solver)

        return bn_solvers, solver, batch_sizes

batch_sizes = [5,10,50]
bn_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('batchnorm')
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```

```
In [13]: plt.subplot(2, 1, 1)
         plot_training_history('Training accuracy (Batch Normalization)','Epoch', solver_bsize
                               lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
         plt.subplot(2, 1, 2)
         plot_training_history('Validation accuracy (Batch Normalization)','Epoch', solver_bsiz
                               lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lab

         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:

위의 그림에 batch size는 batch normalization의 성능에 영향을 주는 경향을 보여주고 있다. 즉 batch size가 작을 수록 성능은 batch size가 큰 모델보다 악화되는 경향을 보여주고 더나아가 baseline model 보다 악화되는 성능을 보여주고 있다. 그 이유는 batch size은 데이터를 샘플링을 하는 경우로 작은 크기는 noisy한 경향을 보여주어 성능이 악화되는 경향을 보여준다.

5 Layer Normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks which have a cap on the input batch size due to hardware limitations.

Several alternatives to batch normalization have been proposed to mitigate this problem; one such technique is Layer Normalization [2]. Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

[2] [Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.](https://arxiv.org/pdf/1607.06450.pdf)

5.1 Inline Question 3:

Which of these data preprocessing steps is analogous to batch normalization, and which is analogous to layer normalization?

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.

3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

5.2 Answer:

2는 layer normalization이고 3이 batch normalization 이다.

6 Layer Normalization: Implementation

Now you'll implement layer normalization. This step should be relatively straightforward, as conceptually the implementation is almost identical to that of batch normalization. One significant difference though is that for layer normalization, we do not keep track of the moving moments, and the testing phase is identical to the training phase, where the mean and variance are directly calculated per datapoint.

Here's what you need to do:

- In `cs231n/layers.py`, implement the forward pass for layer normalization in the function `layernorm_backward`.

Run the cell below to check your results. * In `cs231n/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cs231n/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNet`. When the `normalization` flag is set to `"layernorm"` in the constructor, you should insert a layer normalization layer before each ReLU nonlinearity.

Run the third cell below to run the batch size experiment on layer normalization.

In [14]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after layer normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 =4, 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 layer normalization:')
print_mean_std(a,axis=1)

gamma = np.ones(D3)
beta = np.zeros(D3)
# Means should be close to zero and stds close to one
print('After layer normalization (gamma=1, beta=0)')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)

gamma = np.asarray([3.0,3.0,3.0])
beta = np.asarray([5.0,5.0,5.0])
# Now means should be close to beta and stds close to gamma
print('After layer normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)
```

```
Before layer normalization:
  means:  [-59.06673243 -47.60782686 -43.31137368 -26.40991744]
  stds:   [10.07429373 28.39478981 35.28360729  4.01831507]

After layer normalization (gamma=1, beta=0)
  means:  [ 4.81096644e-16 -7.40148683e-17  2.22044605e-16 -5.92118946e-16]
  stds:   [0.99999995 0.99999999 1.         0.99999969]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
```

```
means:  [5. 5. 5. 5.]
stds:   [2.99999985 2.99999998 2.99999999 2.99999907]
```

```
In [15]: # 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)

         ln_param = {}
         fx = lambda x: layernorm_forward(x, gamma, beta, ln_param)[0]
         fg = lambda a: layernorm_forward(x, a, beta, ln_param)[0]
         fb = lambda b: layernorm_forward(x, gamma, b, ln_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 = layernorm_forward(x, gamma, beta, ln_param)
         dx, dgamma, dbeta = layernorm_backward(dout, cache)

         #You should expect to see relative errors between 1e-12 and 1e-8
         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:  1.4336158494902849e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

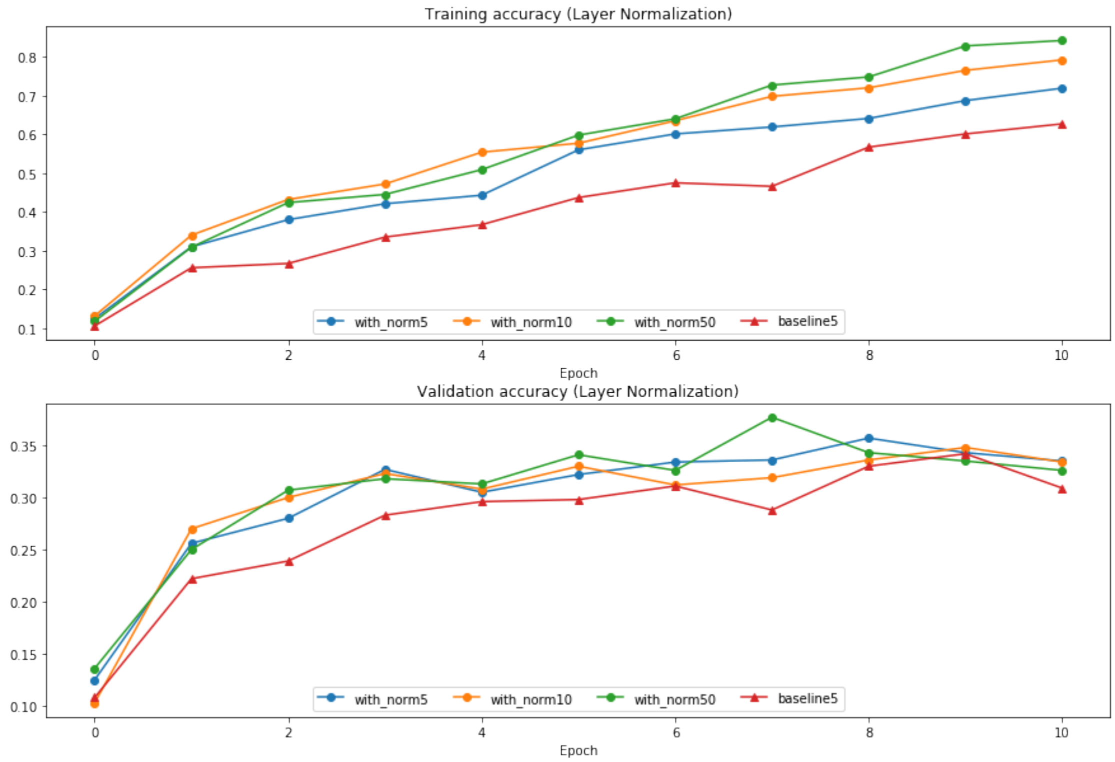
7 Layer Normalization and batch size

We will now run the previous batch size experiment with layer normalization instead of batch normalization. Compared to the previous experiment, you should see a markedly smaller influence of batch size on the training history!

```
In [16]: ln_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('layernorm')

         plt.subplot(2, 1, 1)
         plot_training_history('Training accuracy (Layer Normalization)','Epoch', solver_bsize
                               lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
         plt.subplot(2, 1, 2)
         plot_training_history('Validation accuracy (Layer Normalization)','Epoch', solver_bsi
```

```
                lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lal

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

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```



7.1 Inline Question 4:

When is layer normalization likely to not work well, and why?

1. Using it in a very deep network
2. Having a very small dimension of features
3. Having a high regularization term

7.2 Answer:

layer normalization는 깊은 신경망에서도 잘 작동한다. small size dimenstion은 feature 기반으로 normalization을 하는 layer normalization의 경우 mean과 vairiance가 noisy할 수 있어 성능에 영향을 미친다. high regularization term이 layer normalization을 함께 하는 경우 regularization을 이중으로 모델에 거는 것으로 underfitting 현상이 발생한다.