让深层网络更容易训练，其中一个方法是使用更好的优化器，如SGD+Momentum，这部分我们在FCN这个作业中已经进行了实现；另外一个方法就是改变网络的结构，比如加入BN层。

在实践中，使用了BN的网络对于不好的初始值有更强的鲁棒性，即，该方法减轻了如何合理初始化神经网络这个棘手问题带来的头痛。
该做法是让激活数据在训练开始前通过一个网络，网络处理数据使其服从标准高斯分布（均值为0，方差为1）。

这一次的作业我们就来实现了BN层，然后用它来进行网络训练。
主要内容：
[TOC]

#1 BN层的构建

##1.1 前向传播
前向传播的数学公式如下：

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

 值得注意的是，在做测试的时候为了对一个采样也可以使用BN，此时的均值采用做训练时候的一个统计平均，同时方差采样的是误差的是无差的平均统计，即：

$$u_{test} = E[u]$$

$$var_{test} = \frac{m}{m-1} E[var]$$

根据这个，我们可以确定其在程序上的实现：
```#python
def batchnorm_forward(x, gamma, beta, bn_param):

    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == 'train':
        #############################################################################
        # TODO: Implement the training-time forward pass for batch normalization.   #
        # Use minibatch statistics to compute the mean and variance, use these      #
        # statistics to normalize the incoming data, and scale and shift the        #
        # normalized data using gamma and beta.                                     #
```

```
        #                                                                           #
        # You should store the output in the variable out. Any intermediates that   #
        # you need for the backward pass should be stored in the cache variable.    #
        #                                                                           #
        # You should also use your computed sample mean and variance together with  #
        # the momentum variable to update the running mean and running variance,    #
        # storing your result in the running_mean and running_var variables.        #
        #############################################################################
        sample_mean = np.mean(x, axis=0)
        sample_var = np.var(x, axis=0)
        x_hat = (x - sample_mean) / (np.sqrt(sample_var + eps))
        out = gamma * x_hat + beta
        cache = (gamma, x, sample_mean, sample_var, eps, x_hat)
        running_mean = momentum * running_mean + (1 - momentum) * sample_mean
        running_var = momentum * running_var + (1 - momentum) * sample_var
        # pass
        #############################################################################
        #                             END OF YOUR CODE                              #
        #############################################################################
    elif mode == 'test':
        #############################################################################
        # TODO: Implement the test-time forward pass for batch normalization. Use   #
        # the running mean and variance to normalize the incoming data, then scale  #
        # and shift the normalized data using gamma and beta. Store the result in   #
        # the out variable.                                                         #
        #############################################################################
        scale = gamma / (np.sqrt(running_var + eps))
        out = x * scale + (beta - running_mean * scale)
        # pass
        #############################################################################
        #                             END OF YOUR CODE                              #
        #############################################################################
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    return out, cache
```
>因训练所需而“刻意”加入BN能够有可能还原最初的输入，从而保障整个网络的capacity。

我们来看下训练过程是不是这样。
测试代码如下：
```#python

N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print ('Before batch normalization:')
print ('  means: ', a.mean(axis=0))
```

```
print ('  stds: ', a.std(axis=0))

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

# Now means should be close to beta and stds close to gamma
gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print ('After batch normalization (nontrivial gamma, beta)')
print ('  means: ', a_norm.mean(axis=0))
print ('  stds: ', a_norm.std(axis=0))
```
我们来看下输出结果：
```
Before batch normalization:
means: [ 9.32811104 0.36445057 43.97143731]
stds: [ 30.62687337 39.74610776 39.20258823]
After batch normalization (gamma=1, beta=0)
mean: [ -1.34336986e-16 3.33066907e-17 1.99146255e-17]
std: [ 0.99999999 1. 1. ]
After batch normalization (nontrivial gamma, beta)
means: [ 11. 12. 13.]
stds: [ 0.99999999 1.99999999 2.99999999]
```

X是标准正太分布的输入，经过一层网络之后其分布的均值已经明显不为0，而经过一层BN之后，均值接近于0（方差接近1）。

同样，我们来检验下测试过程：
```#python
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))
```

我们看下输出结果：

```
After batch normalization (test-time):
means: [ 0.11741878 0.10338283 -0.05254307]
stds: [ 1.00233346 0.97693215 1.06506534]
```

尽管我们将训练执行了50次，但是丝毫没有影响我们预测值的分布。

## 1.2 反向传播

反向传播就是对BN的参数更新，公式如下：

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}\right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

代码如下：

```python
def batchnorm_backward(dout, cache):

    dx, dgamma, dbeta = None, None, None
    ###########################################################################
    # TODO: Implement the backward pass for batch normalization. Store the      #
    # results in the dx, dgamma, and dbeta variables.                         #
    ###########################################################################
    gamma, x, u_b, sigma_squared_b, eps, x_hat = cache
    N = x.shape[0]

    dx_1 = gamma * dout
    dx_2_b = np.sum((x - u_b) * dx_1, axis=0)
    dx_2_a = ((sigma_squared_b + eps) ** -0.5) * dx_1
    dx_3_b = (-0.5) * ((sigma_squared_b + eps) ** -1.5) * dx_2_b
    dx_4_b = dx_3_b * 1
    dx_5_b = np.ones_like(x) / N * dx_4_b
    dx_6_b = 2 * (x - u_b) * dx_5_b
    dx_7_a = dx_6_b * 1 + dx_2_a * 1
    dx_7_b = dx_6_b * 1 + dx_2_a * 1
    dx_8_b = -1 * np.sum(dx_7_b, axis=0)
    dx_9_b = np.ones_like(x) / N * dx_8_b
    dx_10 = dx_9_b + dx_7_a

    dgamma = np.sum(x_hat * dout, axis=0)
    dbeta = np.sum(dout, axis=0)
    dx = dx_10
    # pass
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
```

```
    return dx, dgamma, dbeta
```
同样，我们对反向传播的梯度进行检验：
```#python
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, gamma, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
print ('dx error: ', rel_error(dx_num, dx))
print ('dgamma error: ', rel_error(da_num, dgamma))
print ('dbeta error: ', rel_error(db_num, dbeta))
```
得到的误差结果如下：
```
dx error: 1.18652854011e-09
dgamma error: 6.47200668054e-12
dbeta error: 1.67241251033e-10
```

##1.3 反向传播的加速运算
代码如下：
```#python
def batchnorm_backward_alt(dout, cache):

    dx, dgamma, dbeta = None, None, None
    #############################################################################
    # TODO: Implement the backward pass for batch normalization. Store the      #
    # results in the dx, dgamma, and dbeta variables.                           #
    #                                                                           #
    # After computing the gradient with respect to the centered inputs, you     #
    # should be able to compute gradients with respect to the inputs in a       #
    # single statement; our implementation fits on a single 80-character line.  #
    #############################################################################
    gamma, x, sample_mean, sample_var, eps, x_hat = cache
    N = x.shape[0]
    dx_hat = dout * gamma
    dvar = np.sum(dx_hat * (x - sample_mean) * -0.5 * np.power(sample_var + eps, -1.5), axis=0)
    dmean = np.sum(dx_hat * -1 / np.sqrt(sample_var + eps), axis=0) + dvar * np.mean(-2 * (x - sample_mean), axis=0)
    dx = 1 / np.sqrt(sample_var + eps) * dx_hat + dvar * 2.0 / N * (x - sample_mean) + 1.0 / N * dmean
    dgamma = np.sum(x_hat * dout, axis=0)
```

```
    dbeta = np.sum(dout, axis=0)
    # pass
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################

    return dx, dgamma, dbeta
```
我们比较下这两种反向传播的效率和值。
比较代码如下：
```#python
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)))
```
得到的结果如下：
```

dx difference:  2.81820855093e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.01x
```
一层的计算效率比原来高了0.01倍，还是可以接受的，那么接下来我们就可以使用加速后的反向传播计算方法。

这节我们就完成了BN层的前向传播和反向传播的实现。

在正式训练之前，我们还可以做一些小的改进工作。
如果还对我们之前FCN有印象的话，我们是将映射层和激活函数层合为一层，在实现层面，应用这个技巧通常意味着全连接层（或者是卷积层）与激活函数之间添加一个BN层，所以我们也可以将这三层合为一层。
代码如下：
```#python
def affine_bn_relu_forward(x , w , b, gamma, beta, bn_param):
    a, fc_cache = affine_forward(x, w, b)
    bn, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
    out, relu_cache = relu_forward(bn)
    cache = (fc_cache, bn_cache, relu_cache)
```

```
    return out, cache

def affine_bn_relu_backward(dout, cache):
    fc_cache, bn_cache, relu_cache = cache
    dbn = relu_backward(dout, relu_cache)
    da, dgamma, dbeta =  batchnorm_backward_alt(dbn, bn_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db, dgamma, dbeta
```
接下来我们就可以使用BN层进行网络的训练和预测了。

#2 网络训练
使用BN层的代码同上篇文章（第4章），我们已经将其写入```fc_net.py```中，因此这里我们直接进行训练即可。

我们来训练一个六层的网络，并对比下不使用BN层和使用BN层的效率。

训练代码如下：
```#python
data={}
data={'X_train':X_train,'y_train':y_train,'X_val':X_val,'y_val':y_val,'X_test':X_test,'y_test':y_test}
num_train=1000
hidden_dims=[100,100,100,100,100]
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=True)
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=200)
bn_solver.train()

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

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)
```

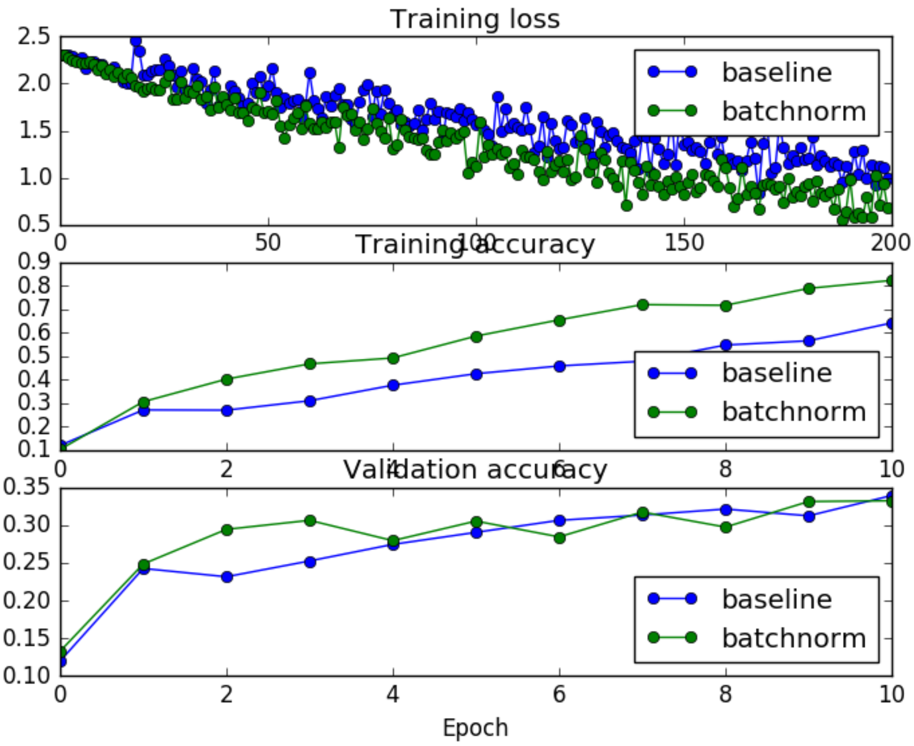
```
plt.plot(solver.loss_history, '-o', label='baseline')
plt.plot(bn_solver.loss_history, '-o', label='batchnorm')
plt.legend(loc='upper right')

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.legend(loc='lower right')

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

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

可视化运行结果如下：




我们可以看下，加入BN层之后，模型可以更快的收敛。（**这是其中一个优点**）

看过cs231n讲义的可能已经知道，不同的参数初始化对我们网络的训练有很大的影响。那么加入了BN层之后，不同的参数初始化，会影响我们的训练吗？
我们来看下。
代码如下：

```python
data={}
data={'X_train':X_train,'y_train':y_train,'X_val':X_val,'y_val':y_val,'X_test':X_test,'y_test':y_test}
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']

}
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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=True)
  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=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

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')
```

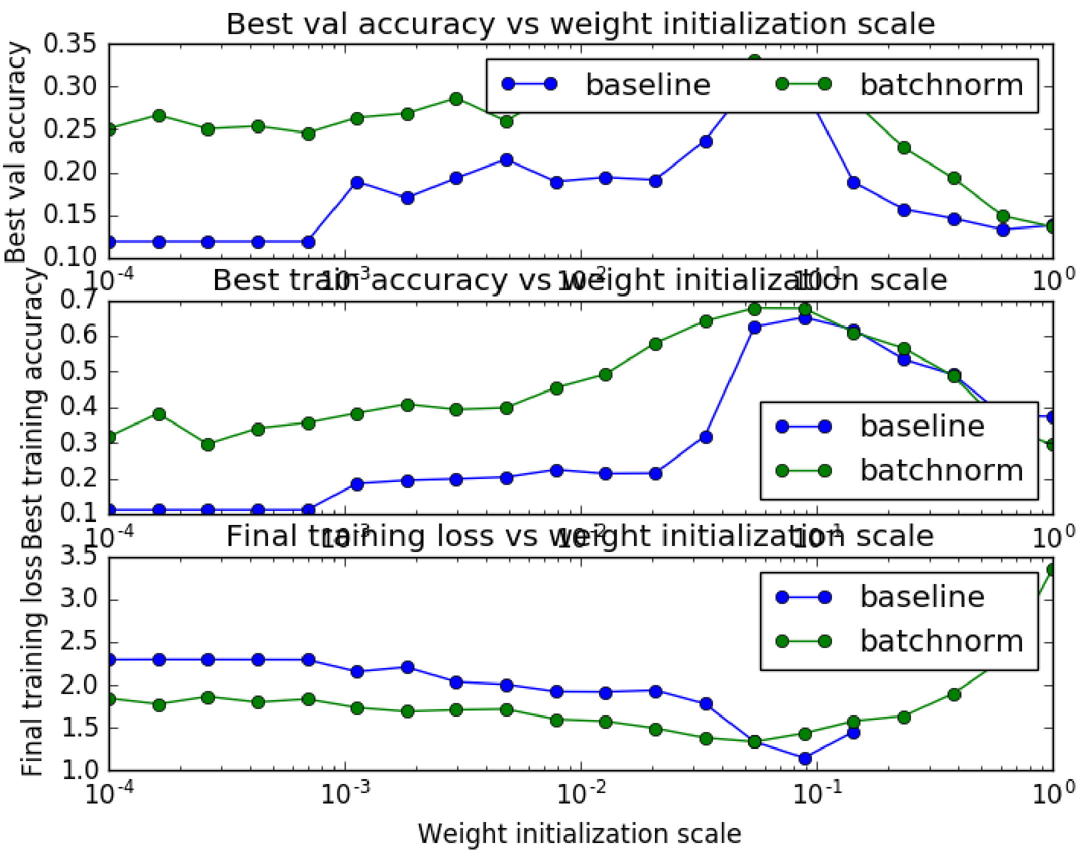
```
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='upper right')

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(loc='lower right')

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()

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

结果如下：




根据可视化结果我们验证了我们刚才的结论，即加入BN层之后可以减少对初始化的强烈依赖。

#3 总结

BN的作用：

1. 改善流经网络的梯度；
2. 允许更大的学习率（学习衰减率也可以很大），大幅提高训练速度；
3. 减少对初始化的强烈依赖；

4. 改善正则化策略：作为正则化的一种形式，轻微减少了对dropout的需求；
5. 再也不需要使用局部响应归一化层了（Alnext中出现），因为BN本身就是一个归一化网络层；
6. 可以把训练数据彻底打乱（防止每批训练的时候，某个样本都经常被挑选到，文献说可以提高1%的精度）。