BatchNormalization

2021 年 5 月 14 日

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.

[2]:
```
# As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from daseCV.classifiers.fc_net import *
from daseCV.data_utils import get_CIFAR10_data
from daseCV.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from daseCV.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()
```

```
run the following from the daseCV directory and try again:
python setup.py build_ext --inplace
```

```
You may also need to restart your iPython kernel
```

[3]:
```python
# 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

在文件 `daseCV/layers` 中实现 `batchnorm_forward` 函数完成 batch normalization 的前向传播。然后运行以下代码测试你的实现是否准确。

上面参考论文 [1] 可能会对你有帮助

[ ]:
```python
# 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]
```

[ ]:
```
# 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:    [1.01531428 1.01238373 0.97819988]
```

1.2 Batch normalization: backward

在 `batchnorm_backward` 中实现 batch normalization 的反向传播

要想得到反向传播的公式，你应该写出 batch normalization 的计算图，并且对每个中间节点求反向传播公式。一些中间节点可能有多个传出分支；注意要在反向传播中对这些分支的梯度求和。

一旦你实现了该功能，请运行下面的代码进行梯度数值检测。

[ ]:
```
# 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)
#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:  1.7029241291468676e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

1.3 Batch normalization: alternative backward

课堂上我们讨论过两种求 sigmoid 反向传播公式的方法，第一种是写出计算图，然后对计算图中的每一个中间变量求导；另一种方法是在纸上计算好最终的梯度，得到一个很简单的公式。打个比方，你可以先在纸上算出 sigmoid 的反向传播公式，然后直接实现就可以了，不需要算中间变量的梯度。

BN 也有这个性质，你可以自己推一波公式！（接下来不翻译了，自己看）

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.

算好之后，在 `batchnorm_backward_alt` 函数中实现简化版的 batch normalization 的反向传播公式，然后分别运行两种反向传播实现并比较结果，你的结果应该是一致的，但是简化版的实现应该会更快一点。

[ ]:
```
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()
```

```python
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:  1.686674680576659e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.62x
```

1.4 Fully Connected Nets with Batch Normalization

现在你已经实现了 Batch Normalization，请在 `daseCV/classifiers/fc_net.py` 中的 `FullyConnectedNet` 上添加 Batch Norm。

具体来说，当在构造函数中 `normalization` 标记设置为 `batchnorm` 时，应该在每个 ReLU 激活层之前插入一个 Batch Norm 层。网络最后一层的输出不应该加 Batch Norm。

当你完成该功能，运行以下代码进行梯度检查。

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

[ ]:
```python
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: 1.33e-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: 1.11e-08
b2 relative error: 4.66e-07
b3 relative error: 2.10e-10
beta1 relative error: 6.65e-09
beta2 relative error: 4.23e-09
gamma1 relative error: 6.27e-09
```

```
gamma2 relative error: 5.28e-09
```

2 Batchnorm for deep networks

运行以下代码，在 1000 个样本的子集上训练一个六层网络，包括有和没有 Batch Norm 的版本。

[4]:
```
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,␣
 ↪normalization='batchnorm')
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()
```

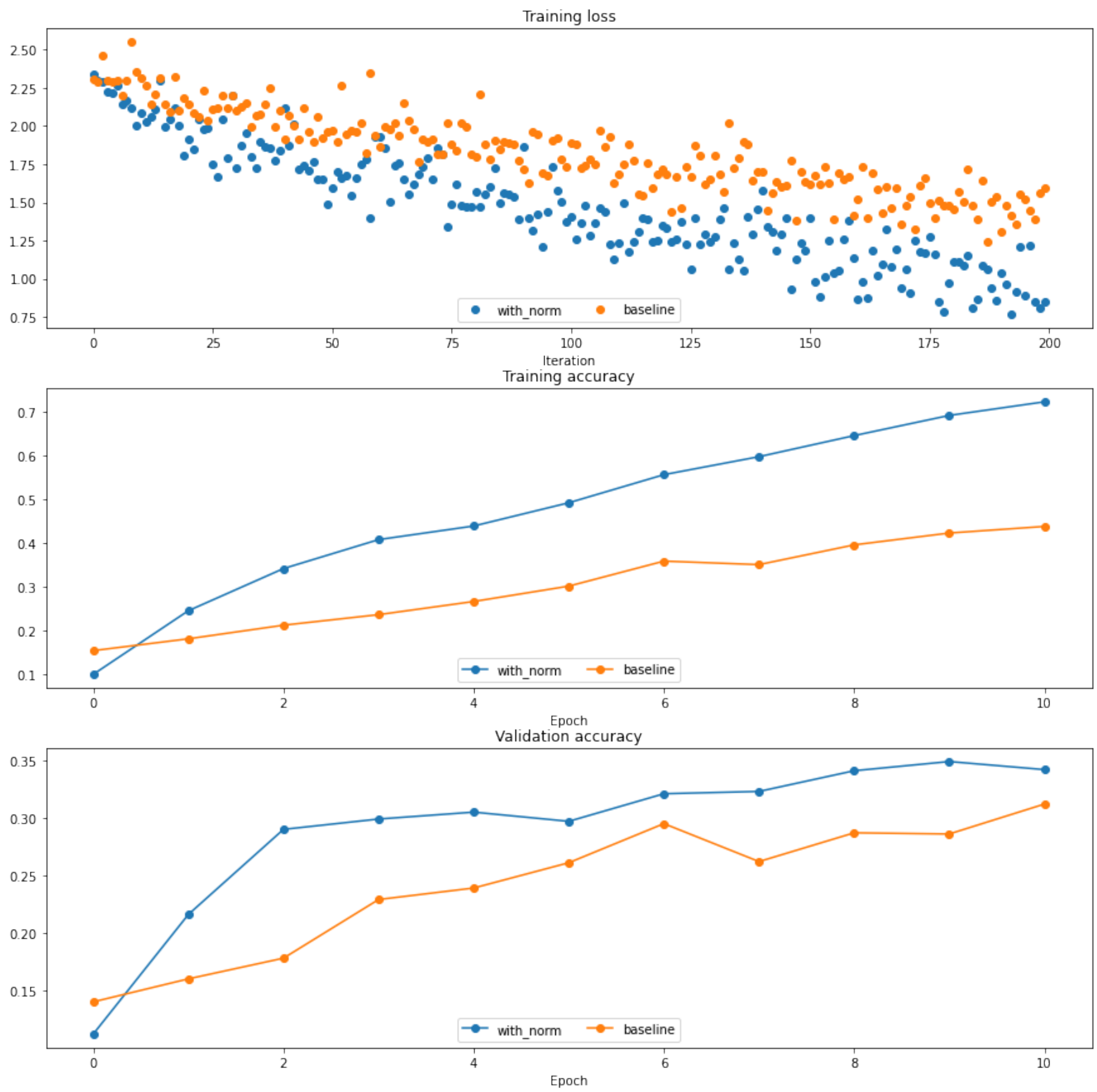
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.101000; val_acc: 0.112000
(Epoch 1 / 10) train acc: 0.246000; val_acc: 0.216000
(Iteration 21 / 200) loss: 1.913054
(Epoch 2 / 10) train acc: 0.342000; val_acc: 0.290000
(Iteration 41 / 200) loss: 2.121798
(Epoch 3 / 10) train acc: 0.408000; val_acc: 0.299000
(Iteration 61 / 200) loss: 1.928388
(Epoch 4 / 10) train acc: 0.439000; val_acc: 0.305000
(Iteration 81 / 200) loss: 1.570334
(Epoch 5 / 10) train acc: 0.492000; val_acc: 0.297000
(Iteration 101 / 200) loss: 1.408623
(Epoch 6 / 10) train acc: 0.556000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.335316
(Epoch 7 / 10) train acc: 0.597000; val_acc: 0.323000
(Iteration 141 / 200) loss: 1.575009
(Epoch 8 / 10) train acc: 0.645000; val_acc: 0.341000
(Iteration 161 / 200) loss: 0.862804
(Epoch 9 / 10) train acc: 0.691000; val_acc: 0.349000
(Iteration 181 / 200) loss: 1.108972
(Epoch 10 / 10) train acc: 0.722000; val_acc: 0.342000

Solver without batch norm:
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.155000; val_acc: 0.140000
(Epoch 1 / 10) train acc: 0.182000; val_acc: 0.160000
(Iteration 21 / 200) loss: 2.143170
```

```
(Epoch 2 / 10) train acc: 0.213000; val_acc: 0.178000
(Iteration 41 / 200) loss: 1.915775
(Epoch 3 / 10) train acc: 0.237000; val_acc: 0.229000
(Iteration 61 / 200) loss: 1.865578
(Epoch 4 / 10) train acc: 0.267000; val_acc: 0.239000
(Iteration 81 / 200) loss: 1.798322
(Epoch 5 / 10) train acc: 0.302000; val_acc: 0.261000
(Iteration 101 / 200) loss: 1.885077
(Epoch 6 / 10) train acc: 0.359000; val_acc: 0.295000
(Iteration 121 / 200) loss: 1.686128
(Epoch 7 / 10) train acc: 0.351000; val_acc: 0.262000
(Iteration 141 / 200) loss: 1.697690
(Epoch 8 / 10) train acc: 0.396000; val_acc: 0.287000
(Iteration 161 / 200) loss: 1.516620
(Epoch 9 / 10) train acc: 0.423000; val_acc: 0.286000
(Iteration 181 / 200) loss: 1.458945
(Epoch 10 / 10) train acc: 0.438000; val_acc: 0.312000
```

运行以下命令来可视化上面训练的两个网络的结果。你会发现，使用 Batch Norm 有助于网络更快地收敛。

[5]:
```
def plot_training_history(title, label, baseline, bn_solvers, plot_fn,␣
↪bl_marker='.', bn_marker='.', labels=None):
    """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

我们将进行一个小实验来研究 Batch Norm 和权值初始化之间的相互关系。

下面代码将训练 8 层网络，分别使用不同规模的权重初始化进行 Batch Norm 和不进行 Batch Norm。然后绘制训练精度、验证集精度、训练损失。

[ ]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
```

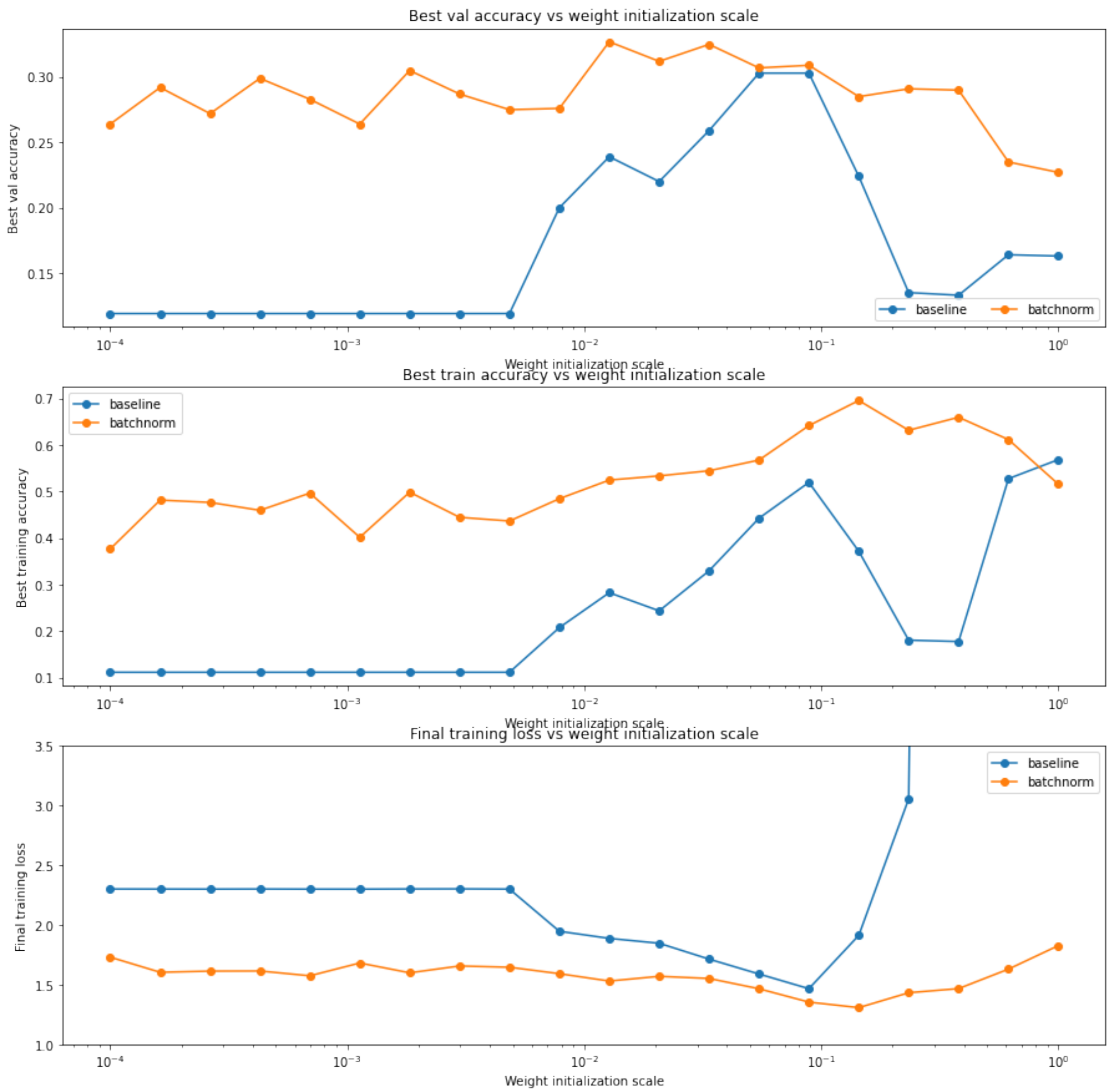
```
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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='batchnorm')
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
↪normalization=None)

  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
```

[ ]:
```
# 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:

描述一下这个实验的结果。权重初始化的规模如何影响带有/没有 Batch Norm 的模型，为什么？

3.2 Answer:

当权重初始化较大时,由于 baseline 依然逐个 normalization,导致训练更不可控,而 batchnorm 依然能保持和小权重时差不多的 loss 水平，因为 batch 后不易受受个体影响。同时可以看出，batchnorm

整体由于 basenorm，这是因为深层神经网络内部特征的转移分布可能会使训练深层网络更加困难，而 batchnorm 则解决了这一点。

4 Batch normalization and batch size

我们将进行一个小实验来研究 Batch Norm 和 batch size 之间的相互关系。

下面的代码将使用不同的 batch size 来训练带有/没有 Batch Norm 的 6 层网络。然后将绘制随时间变化的训练准确率和验证集的准确率。

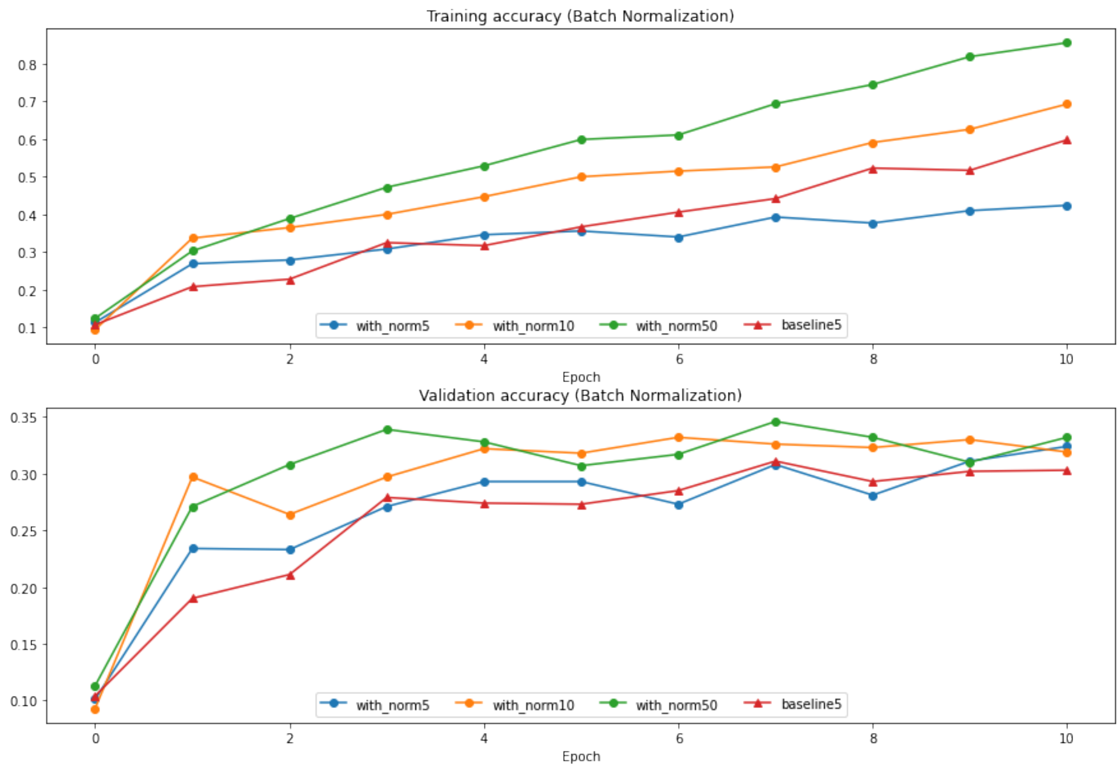
[ ]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(231)
    # Try training a very deep net with batchnorm
    hidden_dims = [100, 100, 100, 100, 100]
    num_train = 1000
    small_data = {
      'X_train': data['X_train'][:num_train],
      'y_train': data['y_train'][:num_train],
      '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=None)
    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,␣
 ↪normalization=normalization_mode)
        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
```

[ ]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
```

```
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',␣
↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
↪bn_marker='-o', labels=batch_sizes)

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



4.1 Inline Question 2:

描述一下这个实验的结果。请问 Batch Norm 和 batch size 之间的又什么关系？为什么会出现这种关系？

4.2 Answer:

batch size 越大时，batch norm 的模型准确率更高，因为这样会有更快的收敛速度和并行能力。只要硬件资源足够，并且模型没明显过拟合，尽量把 batch size 设大。

5 Layer Normalization

（这里大概讲的是 batch norm 受限于 batch size 的取值，但是受限于硬件资源，batch size 不能取太大，所以提出了 layer norm，对一个样本的特征向量进行归一化，均值和方差由该样本的特征向量的所有元素算出来，具体的自己看英文和论文。）

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.

5.1 Inline Question 3:

下面的数据预处理步骤中，哪些类似于 Batch Norm，哪些类似于 Layer Norm?

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:

Batch norm 是按样本批量，Layer Norm 按通道/时间片 (RNN)/隐层节点数量 (MLP) 批量按特征数做 norm. 1.Batch norm，对样本批量 2.Layer Norm，因为是通道数 3.Batch norm，对样本进行的均值化 4.Layerorm，对通道值均值化。

6 Layer Normalization: Implementation

现在你要实现 layer normalization。这步应该相对简单，因为在概念上，layer norm 的实现几乎与 batch norm 一样。不过一个重要的区别是，对于 layer norm，我们使用 moments，并且测试阶段与训练阶段是相同的，每个数据样本直接计算平均值和方差。

你要完成下面的工作

- 实现 `daseCV/layers.py` 中的 `layernorm_forward`。

运行下面第一个 cell 检查你的结果

- 实现 `daseCV/layers.py` 中的 `layernorm_backward`。运行下面第二个 cell 检查你的结果
- 修改 `daseCV/classifiers/fc_net.py`，在 `FullyConnectedNet` 上增加 layer normalization。当构造函数中的 `normalization` 标记为`"layernorm"`时，你应该在每个 ReLU 层前插入 layer normalization 层。

运行下面第三个 cell 进行关于在 layer normalization 上的 batch size 的实验。

[ ]:
```
# 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]
```

[ ]:
```
# 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:  2.107277051835058e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.5842537629899423e-12
```

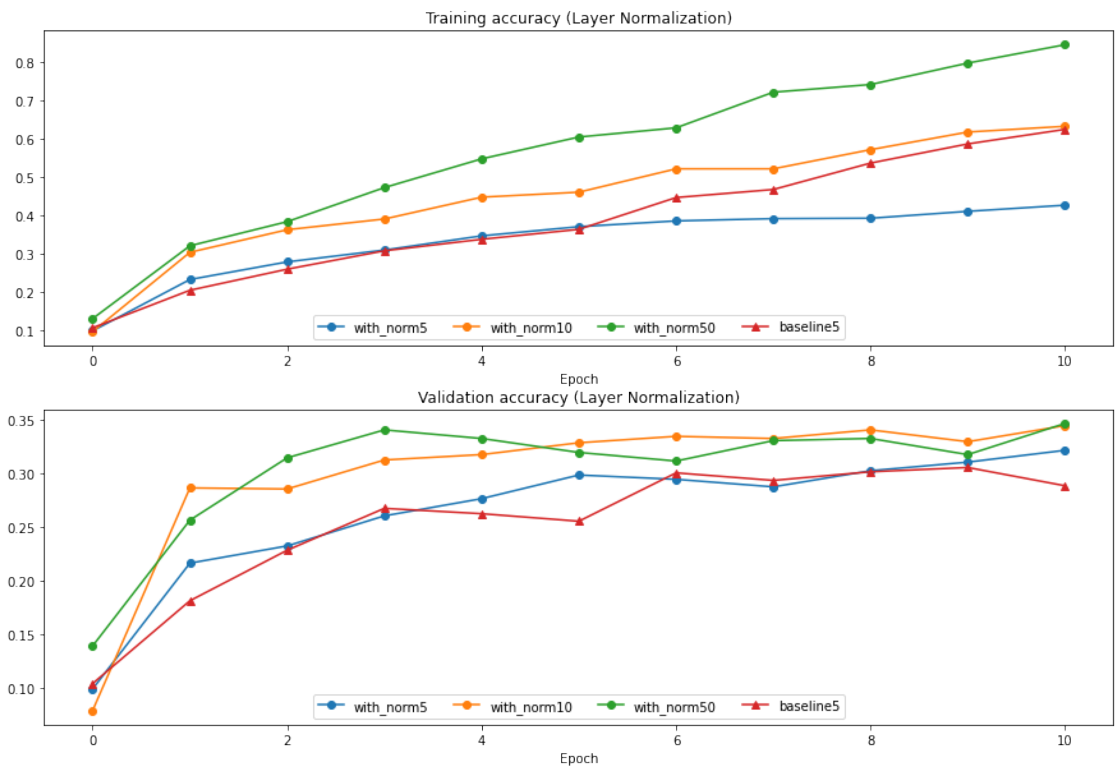
7 Layer Normalization and batch size

我们将使用 layer norm 来进行前面的 batch size 实验。与之前的实验相比，batch size 对训练精度的影响要小得多!

[ ]:
```
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, ln_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
```

```
plot_training_history('Validation accuracy (Layer Normalization)','Epoch',␣
↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
↪bn_marker='-o', labels=batch_sizes)

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:

什么时候 layer normalization 可能不工作（不起作用），为什么？

1. 在非常深的网络上使用
2. 特征的维度非常的小
3. 有非常高的正则化项

7.2 Answer:

2 LN 根据样本的特征数做归一化, 样本特征小了 LN 不起作用。

---

8 重要

这里是作业的结尾处，请执行以下步骤:

1. 点击 `File -> Save` 或者用 `control+s` 组合键，确保你最新的的 notebook 的作业已经保存到谷歌云。
2. 执行以下代码确保 `.py` 文件保存回你的谷歌云。

[6]:
```
import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['daseCV/classifiers/cnn.py', 'daseCV/classifiers/fc_net.py',\
                 'daseCV/layers.py','daseCV/optim.py',\
                 'daseCV/solver.py','daseCV/layer_utils.py',\
                 'daseCV/data_utils.py','daseCV/fast_layers.py']

for files in FILES_TO_SAVE:
  with open(os.path.join(FOLDER_TO_SAVE, '/'.join(files.split('/')[1:])), 'w')␣
↪as f:
    f.write(''.join(open(files).readlines()))
```