In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
!jupyter nbconvert --to html ./BatchNormalization.ipynb
```

In [ ]:
```
# This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzippec
# assignment folder, e.g. 'cs6353/assignments/assignment3/'
# FOLDERNAME = 'assignment3'
# assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall\ 2024/CS\ 6353/CS
!bash get_datasets.sh
%cd ../..

# Install requirements from colab_requirements.txt
# TODO: Please change your path below to the colab_requirements.txt file
# ! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/requiremen
```

```
/Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall 2024/CS 6353/CS_6353/
assignment3/cs6353/datasets
--2024-10-20 15:42:49--  https://www.cs.toronto.edu/~kriz/cifar-10-python.ta
r.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... c
onnected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  45.4MB/s    in 4.0s

2024-10-20 15:42:53 (40.7 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/
170498071]

x cifar-10-batches-py/
x cifar-10-batches-py/data_batch_4
x cifar-10-batches-py/readme.html
x cifar-10-batches-py/test_batch
x cifar-10-batches-py/data_batch_3
x cifar-10-batches-py/batches.meta
x cifar-10-batches-py/data_batch_2
x cifar-10-batches-py/data_batch_5
x cifar-10-batches-py/data_batch_1
/Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall 2024/CS 6353/CS_6353/
assignment3
```

Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, and RMSProp. 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 [3] 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, [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.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

In [ ]:
```
# As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gr
from cs6353.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'

%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 [ ]:
```
# 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,)
```

Batch Normalization: Forward

In the file `cs6353/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above would be helpful!

In [ ]:
```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:  [8.88178420e-18 2.27595720e-17 5.30825384e-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 [ ]:
```
# 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]
```

Batch normalization: Backward Pass

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 [ ]:
```
# 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:  1.0906010595521823e-09
dgamma error:  5.418458160170129e-12
dbeta error:  2.276445013433725e-12
```

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.

Given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_N \end{bmatrix}$, we first calculate the mean $\mu = \frac{1}{N}\sum_{k=1}^{N} x_k$ and variance $v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2$.
With μ and v calculated, we can calculate the standard deviation $\sigma = \sqrt{v + \epsilon}$ and normalized data Y with $y_i = \frac{x_i - \mu}{\sigma}$.

The meat of our problem is to get $\frac{\partial L}{\partial X}$ from the upstream gradient $\frac{\partial L}{\partial Y}$. It might be challenging to directly reason about the gradients over X and Y - try reasoning about it in terms of x_i and y_i first.

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 steps are all as simple as possible.

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 [ ]:
```
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:  5.645352874054582e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 2.14x
```

Fully Connected Networks with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs6353/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 `cs6353/layer_utils.py`.

In [ ]:
```
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,
    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: 4.03e-06
W3 relative error: 2.97e-10
b1 relative error: 2.22e-08
b2 relative error: 2.22e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.85e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 6.96e-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: 4.44e-08
b2 relative error: 2.22e-08
b3 relative error: 2.23e-10
beta1 relative error: 6.32e-09
beta2 relative error: 5.69e-09
gamma1 relative error: 5.94e-09
gamma2 relative error: 4.14e-09
```

Batch Normalization for Deep Networks

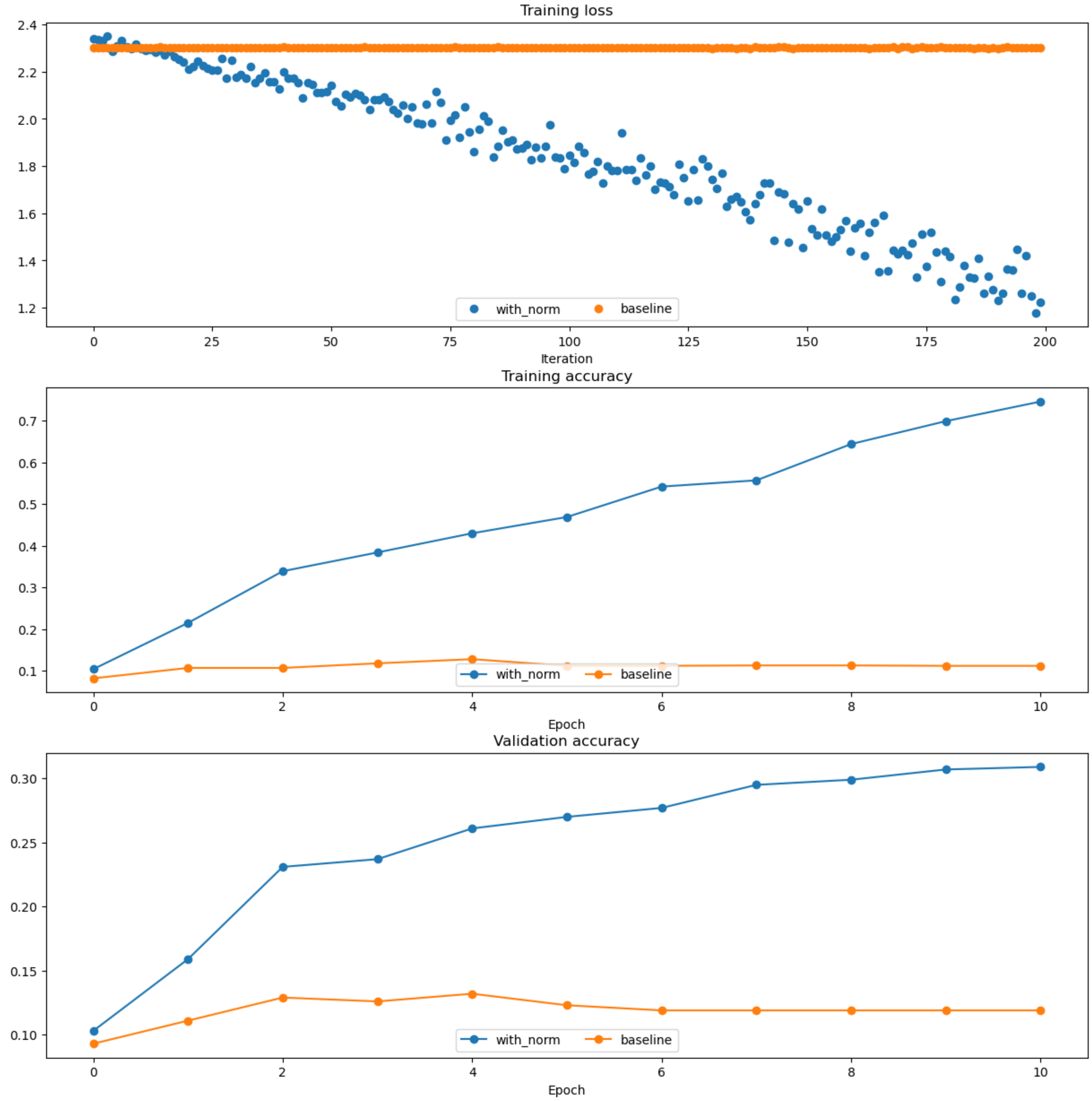
Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

In [ ]:
```
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, normali
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalizat

bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='sgd_momentum',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True,print_every=20)
bn_solver.train()

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='sgd_momentum',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.104000; val_acc: 0.103000
(Epoch 1 / 10) train acc: 0.215000; val_acc: 0.159000
(Iteration 21 / 200) loss: 2.210432
(Epoch 2 / 10) train acc: 0.339000; val_acc: 0.231000
(Iteration 41 / 200) loss: 2.200850
(Epoch 3 / 10) train acc: 0.384000; val_acc: 0.237000
(Iteration 61 / 200) loss: 2.080494
(Epoch 4 / 10) train acc: 0.430000; val_acc: 0.261000
(Iteration 81 / 200) loss: 1.860840
(Epoch 5 / 10) train acc: 0.469000; val_acc: 0.270000
(Iteration 101 / 200) loss: 1.844514
(Epoch 6 / 10) train acc: 0.542000; val_acc: 0.277000
(Iteration 121 / 200) loss: 1.728642
(Epoch 7 / 10) train acc: 0.557000; val_acc: 0.295000
(Iteration 141 / 200) loss: 1.678778
(Epoch 8 / 10) train acc: 0.644000; val_acc: 0.299000
(Iteration 161 / 200) loss: 1.539485
(Epoch 9 / 10) train acc: 0.699000; val_acc: 0.307000
(Iteration 181 / 200) loss: 1.416017
(Epoch 10 / 10) train acc: 0.746000; val_acc: 0.309000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.082000; val_acc: 0.093000
(Epoch 1 / 10) train acc: 0.107000; val_acc: 0.111000
(Iteration 21 / 200) loss: 2.302091
(Epoch 2 / 10) train acc: 0.107000; val_acc: 0.129000
(Iteration 41 / 200) loss: 2.303817
(Epoch 3 / 10) train acc: 0.118000; val_acc: 0.126000
(Iteration 61 / 200) loss: 2.300032
(Epoch 4 / 10) train acc: 0.128000; val_acc: 0.132000
(Iteration 81 / 200) loss: 2.302165
(Epoch 5 / 10) train acc: 0.112000; val_acc: 0.123000
(Iteration 101 / 200) loss: 2.302931
(Epoch 6 / 10) train acc: 0.112000; val_acc: 0.119000
(Iteration 121 / 200) loss: 2.300168
(Epoch 7 / 10) train acc: 0.113000; val_acc: 0.119000
(Iteration 141 / 200) loss: 2.302988
(Epoch 8 / 10) train acc: 0.113000; val_acc: 0.119000
(Iteration 161 / 200) loss: 2.299930
(Epoch 9 / 10) train acc: 0.112000; val_acc: 0.119000
(Iteration 181 / 200) loss: 2.299801
(Epoch 10 / 10) train acc: 0.112000; val_acc: 0.119000
```

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 [ ]:
```
def plot_training_history(title, label, baseline, bn_solvers, plot_fn, bl_ma
    """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_mark
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

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


Training loss
with_norm
baseline
Iteration
Training accuracy
with_norm
baseline
Epoch
Validation accuracy
with_norm
baseline
Epoch

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 [ ]:
```
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, norma
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normaliz

  bn_solver = Solver(bn_model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='sgd_momentum',
                  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='sgd_momentum',
                  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
```

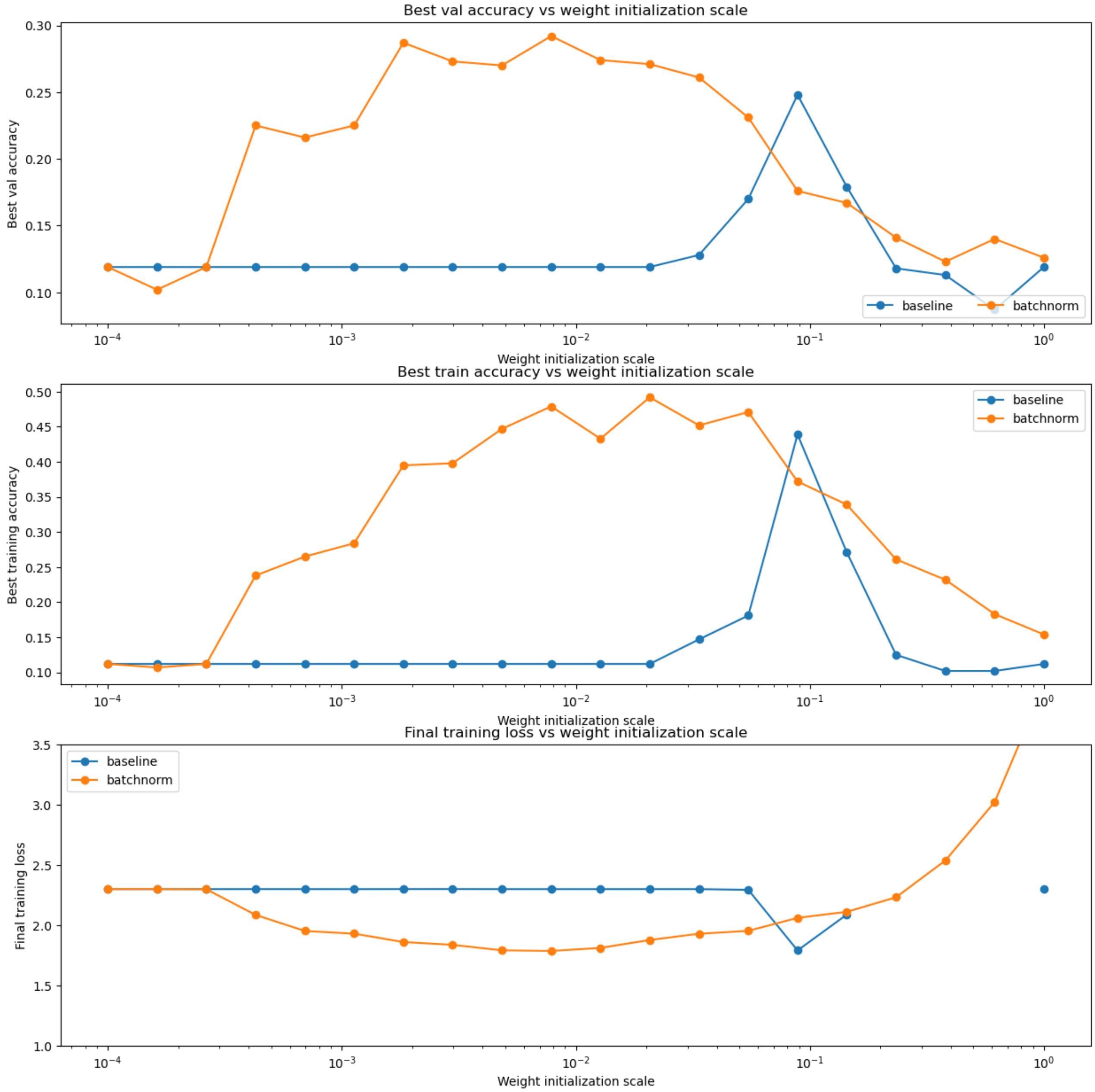
```
/Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall 2024/CS 6353/CS_6353/
assignment3/cs6353/classifiers/fc_net.py:323: RuntimeWarning: overflow encou
ntered in multiply
  loss += 0.5 * reg * (np.sum(w[key] * w[key]))
/Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall 2024/CS 6353/CS_6353/
assignment3/cs6353/classifiers/fc_net.py:323: RuntimeWarning: invalid value
encountered in scalar multiply
  loss += 0.5 * reg * (np.sum(w[key] * w[key]))
/opt/anaconda3/envs/cs6353/lib/python3.11/site-packages/numpy/core/fromnumer
ic.py:88: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```

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

```
/Users/novellaalvina/Documents/US/UTAH/Lessons/MS/Fall 2024/CS 6353/CS_6353/
assignment3/cs6353/layers.py:423: RuntimeWarning: invalid value encountered
in subtract
  shifted_logits = x - np.max(x, axis=1, keepdims=True)
```

```
Running weight scale 20 / 20
```

In [ ]:
```
# 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()
```



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?

Answer:

Based on the figures above, it is shown that there has not been much improvement in the training and validation accuracy until 10-2 weight initialization scale for the baseline model. In contrast with batchnorm model, that the training and validation accuracy increases even from around 10-4.

This indicates that the batch normalization significantly improves the stability in training and generalization of the model ability. This, in turn, allows the model to achieve high accuracy across a wide range of inintiallization scales.

Whereas the baseline model has the stabilizing and improving training normalization converge, which is the result of the model too heavily reliant on the proper choice of weight initialization scale. Hence, it struggles to reach the effective learning.

Hence, Batch Normalization is more recommended since it makes the training process becomes more robust and less dependent the proper choice of weight initialization scale.

Batch normalization and batch size

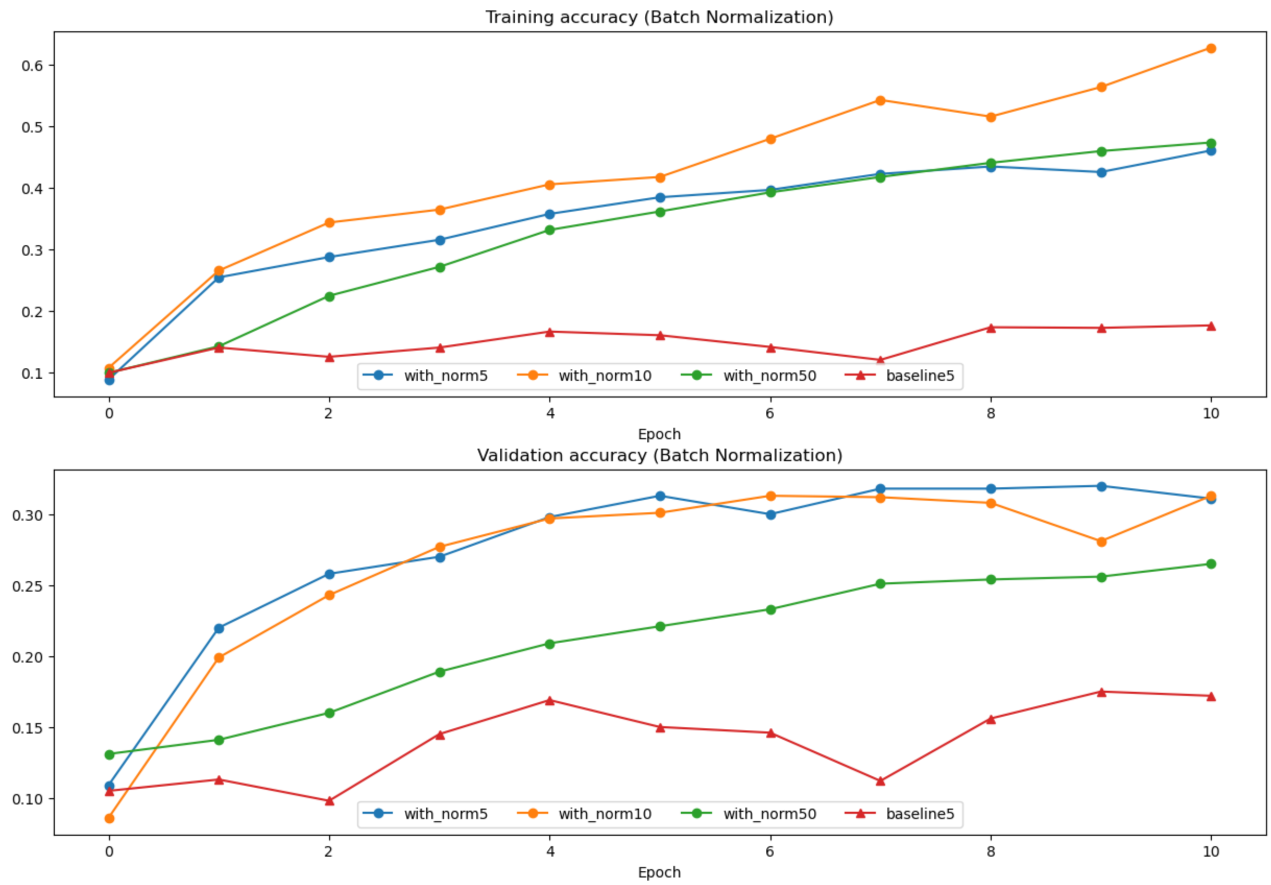
We will now run a small experiment to study the interaction of batch normalization and batch size.

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 [ ]:
```
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, normal
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='sgd_momentum',
                    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,
        bn_solver = Solver(bn_model, small_data,
                        num_epochs=n_epochs, batch_size=b_size,
                        update_rule='sgd_momentum',
                        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('bat
```

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

In [ ]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch', sol
                      lambda x: x.train_acc_history, bl_marker='-^', bn_mark
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch', s
                      lambda x: x.val_acc_history, bl_marker='-^', bn_marker

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



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?

Answer:

Based on the produced figures above, with_norm5 (blue) and with_norm10 (orange) perform better compared to with_norm50 (green) and the baseline, both in terms of training and validation accuracy.

This implies batch normalization is more effective with smaller to moderate batch sizes (e.g., 5 or 10), as the noisier batch statistics provide useful regularization that aids generalization. For larger batch sizes (e.g., 50), the reduced variability of the noise in batch statistics diminishes the regularization effect, leading to overfitting and poorer generalization.

The baseline model without batch normalization shows very poor performance, highlighting the role of batch normalization in stabilizing training and enhancing both learning and generalization. These observations suggest that while batch normalization is generally beneficial, the choice of batch size can significantly influence its effectiveness, with smaller batch sizes often leading to better generalization due to the added regularization effect.

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.

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.

Answer:

Batch Normalization: Given multiple feature vectors (samples), each distinct feature (at the same index) is normalized across multiple samples.

Layer Normalization: Given a single feature vector, each distinct feature is normalized based on other features (at different indices) across that same vector.

Step 2: Scaling each image so that the RGB channels for all pixels within an image sums up to 1. This step involves normalizing all pixels within a single image. It is analogous to Layer Normalization, as it normalizes across all features (pixels) of a single image independently, similar to how LN normalizes within each sample.

Step 3: Subtracting the mean image of the dataset from each image in the dataset. This step involves computing a mean image across all images in the dataset and then subtracting this mean from each individual image. It is analogous to Batch Normalization: Subtracting the mean image from all images uses batch-level statistics, akin to BN using batch-level normalization.

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 `cs6353/layers.py`, implement the forward pass for layer normalization in the function `layernorm_backward`.

Run the cell below to check your results.

- In `cs6353/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results.

- Modify `cs6353/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 [ ]:
```
# 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 -1.48029737e-16 -2.59052039e-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 [ ]:
```
# 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))
```

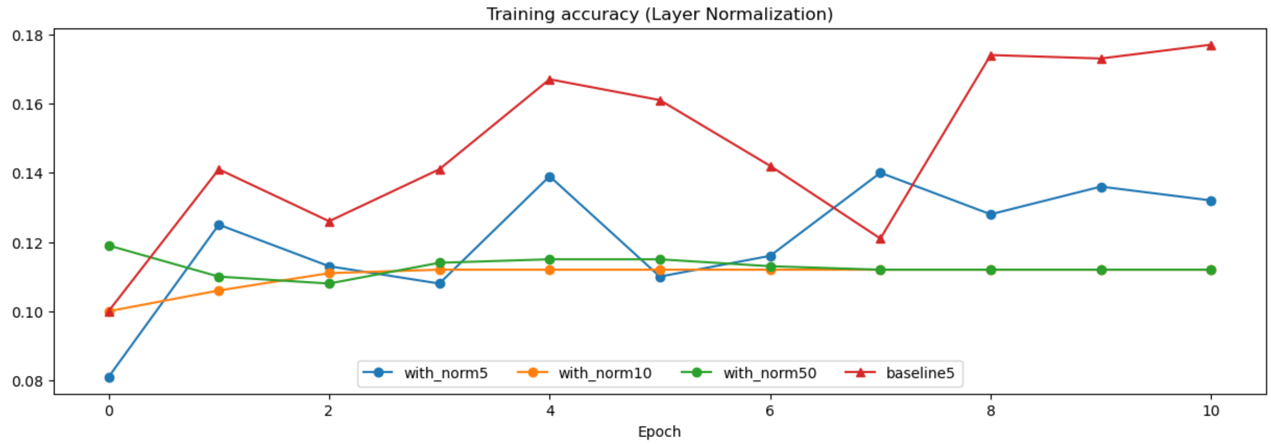
```
dx error:  9.469526479480692e-10
dgamma error:  1.9793843388564062e-12
dbeta error:  2.276445013433725e-12
```

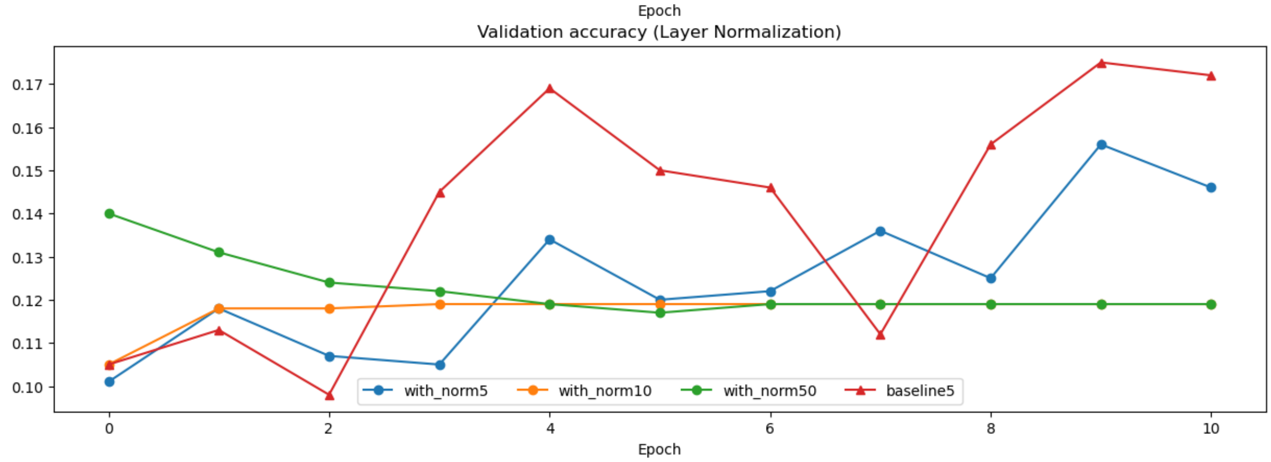
Layer Normalization and batch size

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 [ ]:
```
ln_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('lay

plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Layer Normalization)','Epoch', sol
                      lambda x: x.train_acc_history, bl_marker='-^', bn_mark
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Layer Normalization)','Epoch', s
                      lambda x: x.val_acc_history, bl_marker='-^', bn_marker

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
```





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

Answer:

1. **FALSE**: Since feature vectors are normalized per layer, adding more layers will not reduce the performance of Layer Normalization. Normalizing each feature vector roughly centers it, resulting in a less skewed loss function topology. Moreover, scaling each feature balances extreme values (e.g., very high or very low), preventing gradients from vanishing or exploding.

2. **TRUE**: When there are only a few values in the feature vector, it becomes difficult to accurately estimate the mean and variance, leading to poor scaling across features. This can cause inconsistent performance.

3. **TRUE**: High regularization may penalize weights too much, preventing them from adequately emphasizing specific features. In general, a high regularization term simplifies the model and tends to increase the loss.