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 unzipped
# 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 [ ]:
```
# 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 $X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}$, 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

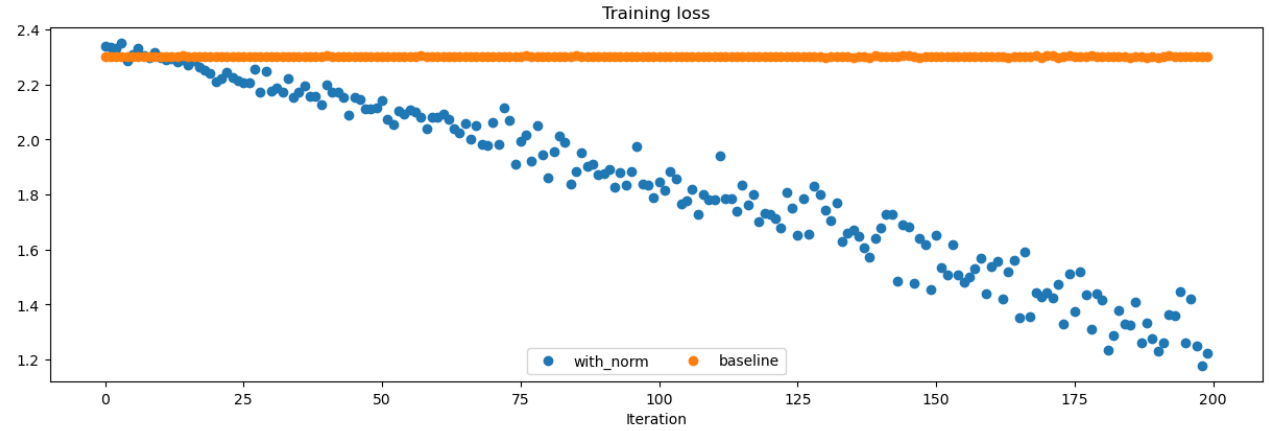
Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

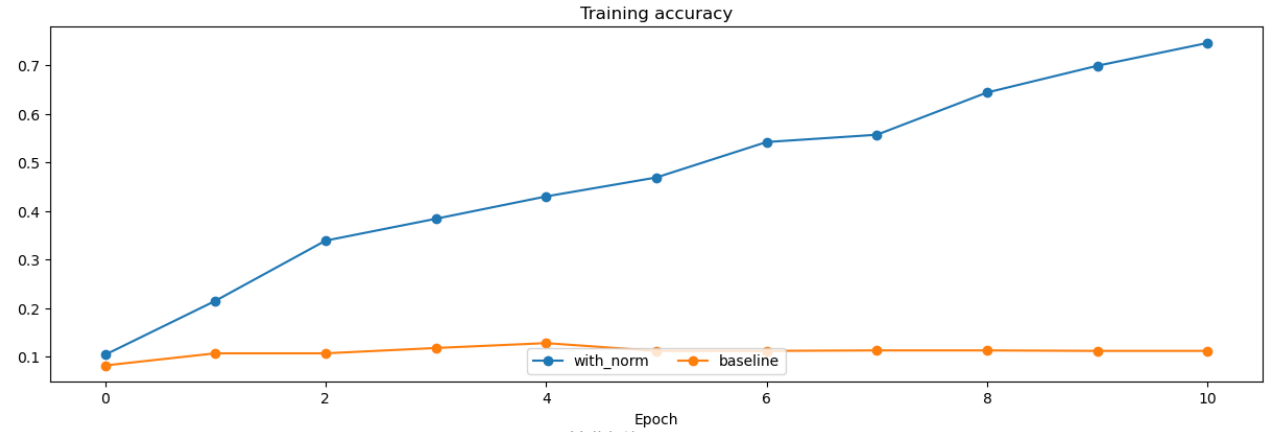
In [ ]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

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()
```

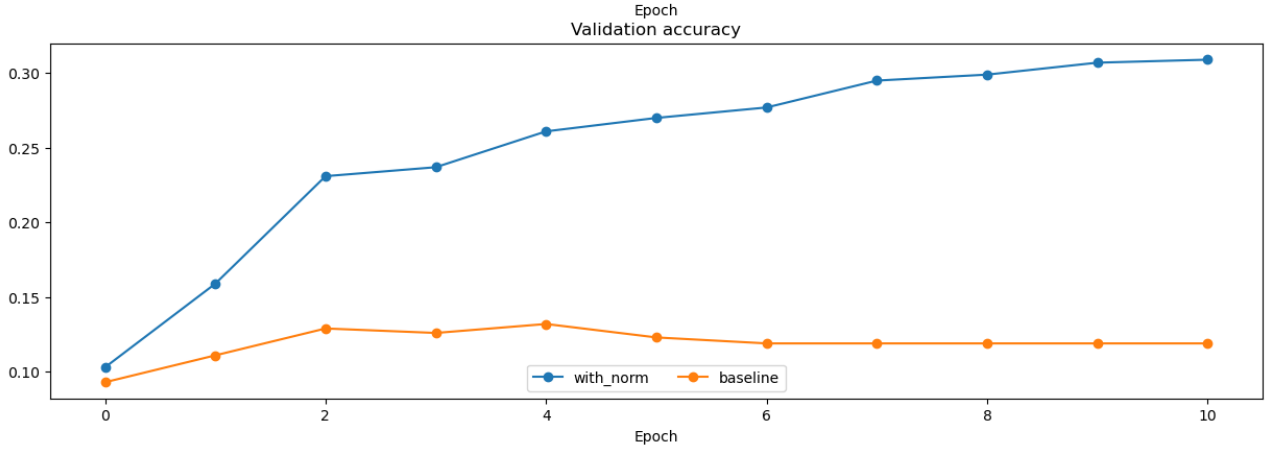
```
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.104000; val_acc: 0.103000
(Epoch 1 / 10) train acc: 0.215000; val_acc: 0.159000
(Iteration 21 / 200) loss: 2.210432
(Epoch 2 / 10) train acc: 0.339000; val_acc: 0.231000
(Iteration 41 / 200) loss: 2.200850
(Epoch 3 / 10) train acc: 0.384000; 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
2.4
2.2
2.0
1.8
1.6
1.4
1.2
with_norm
baseline
0
25
50
75
100
125
150
175
200
Iteration


Training accuracy
0.7
0.6
0.5
0.4
0.3
0.2
0.1
with_norm
baseline
0
2
4
6
8
10
Epoch


Validation accuracy
0.30
0.25
0.20
0.15
0.10
with_norm
baseline
0
2
4
6
8
10
Epoch

Batch Normalization and Initialization

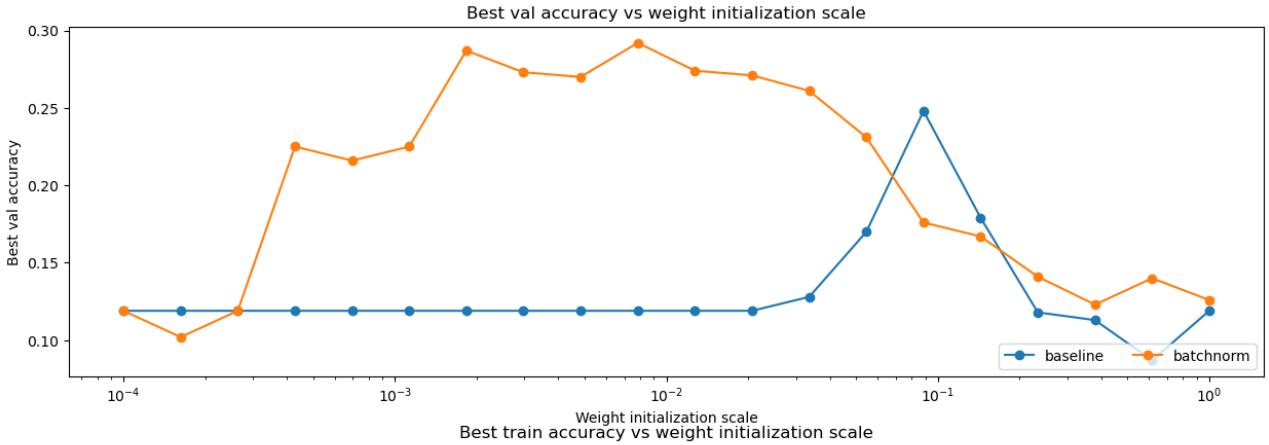
We will now run a small experiment to study the interaction of batch normalization and weight initialization.

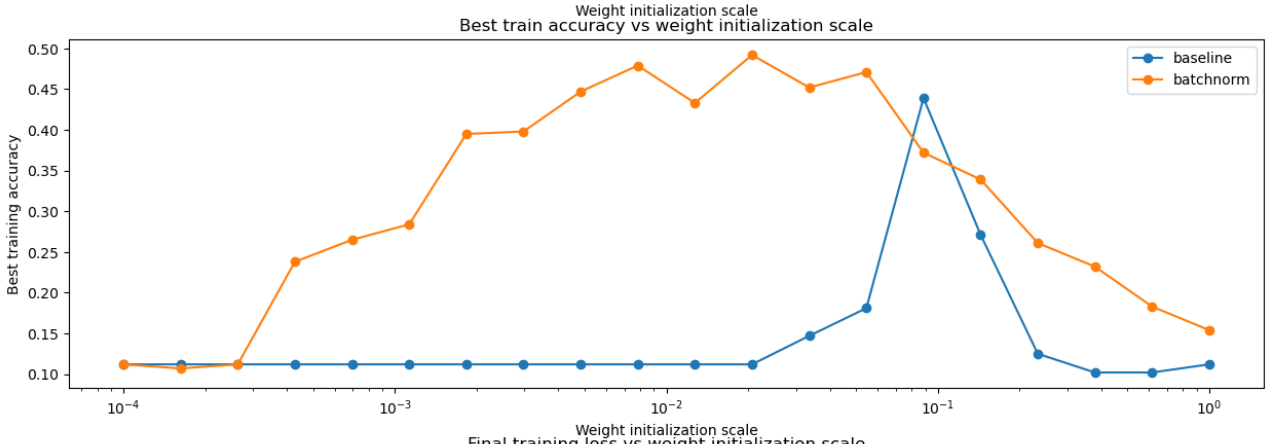
The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

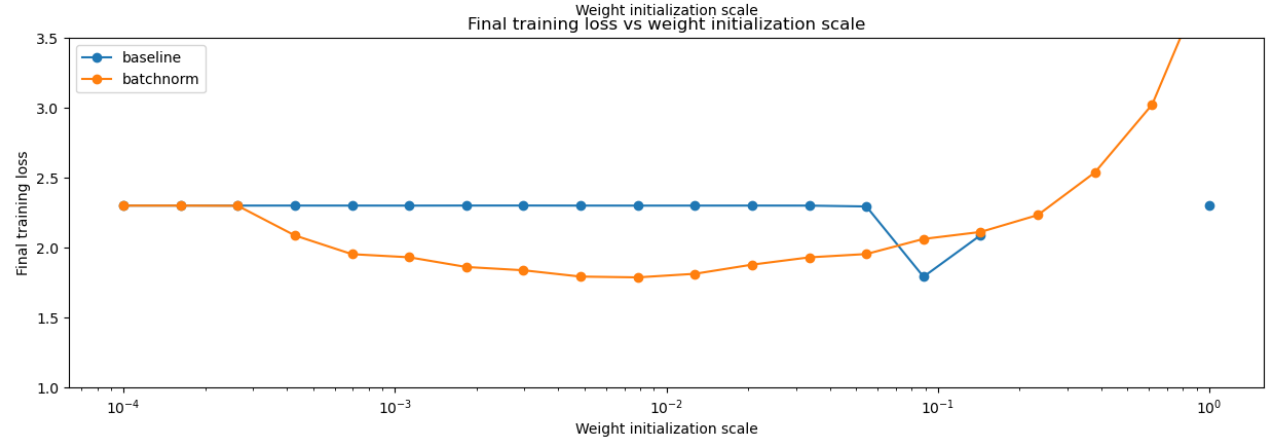
In [ ]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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

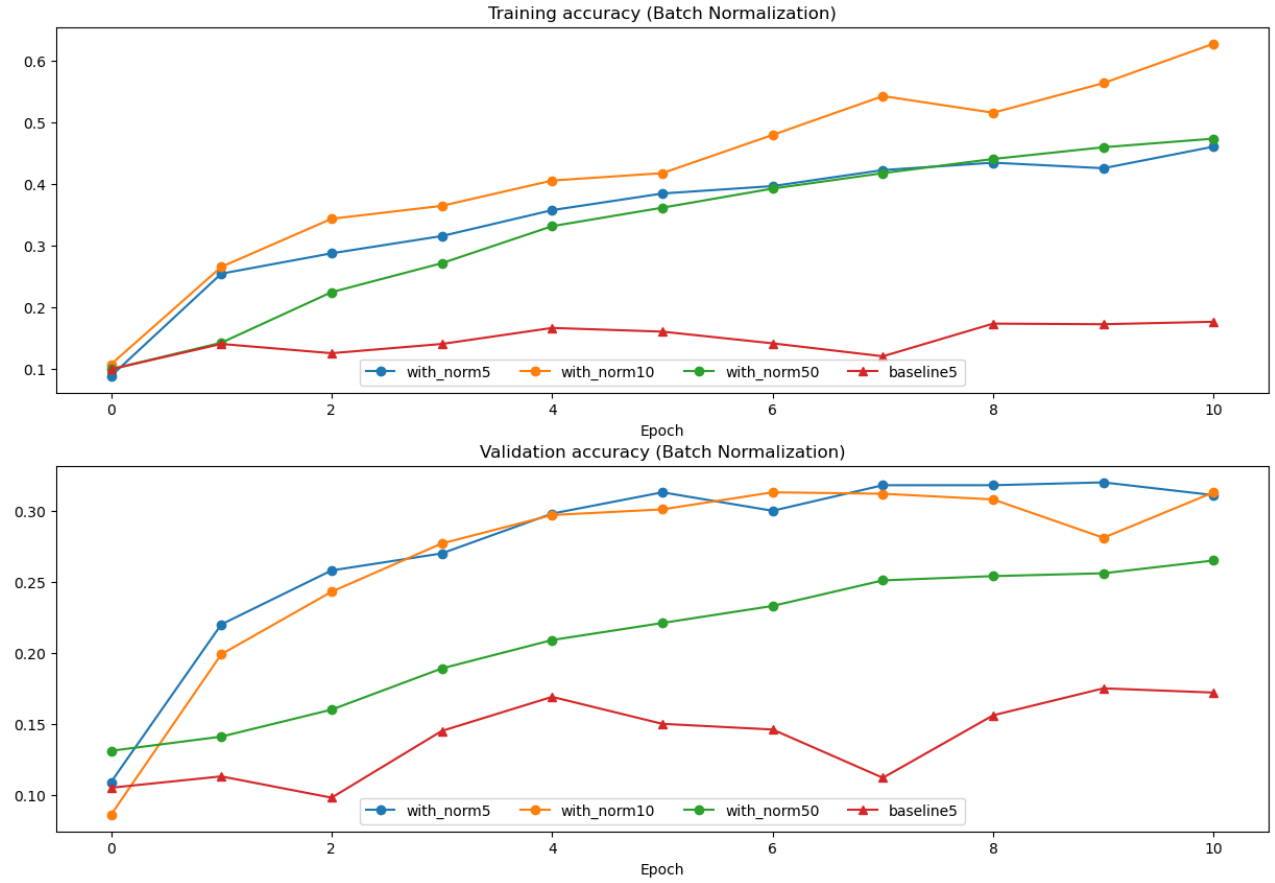
We will now run a small experiment to study the interaction of batch normalization and batch size.

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))
```

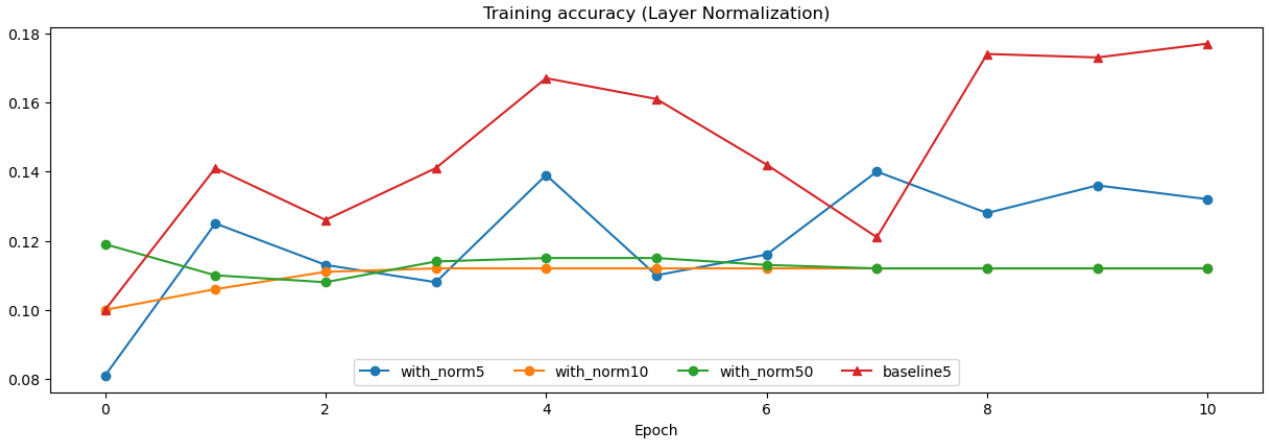
```
dx error:  9.469526479480692e-10
dgamma error:  1.9793843388564062e-12
dbeta error:  2.276445013433725e-12
```

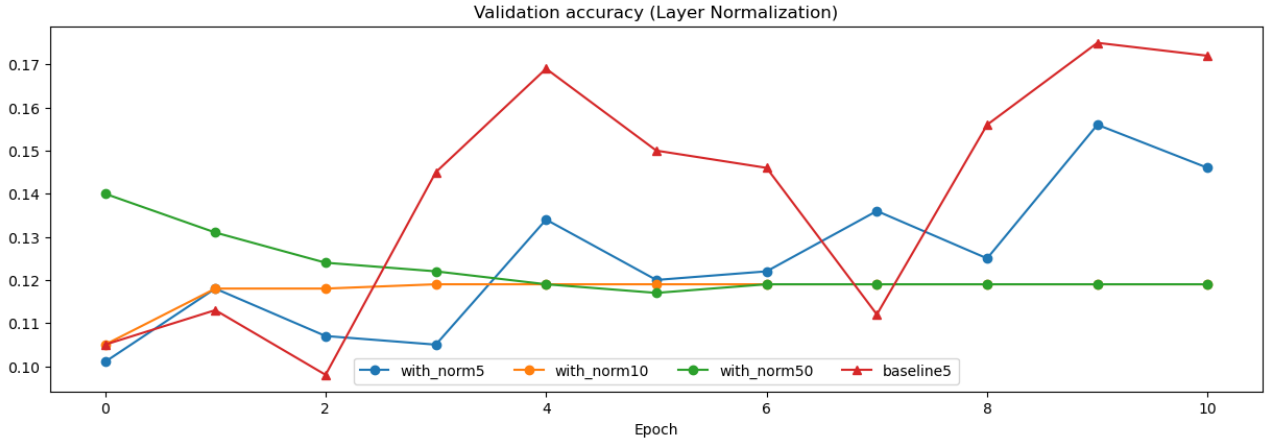
Layer Normalization and batch size

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.

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and
cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w
```

```
    return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

In [1]:
```
# 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 unzipped
# 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 /content/drive/MyDrive/MS/CS6353/assignment3/cs6353/datasets
!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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, cal
l drive.mount("/content/drive", force_remount=True).
/content/drive/MyDrive/MS/CS6353/assignment3/cs6353/datasets
--2024-10-21 03:57:42--  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  25.0MB/s    in 6.6s

2024-10-21 03:57:50 (24.8 MB/s) - 'cifar-10-python.tar.gz' saved [170498071/
170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/MyDrive/MS/CS6353/assignment3
```

In [2]:
```
# As usual, a bit of setup
from __future__ import print_function
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'

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

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

In [3]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k, v in list(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,))
```

Affine layer: foward

Open the file `cs6353/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

In [4]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), ou
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)

correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

In [5]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0],
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0],
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0],

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

In [6]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,
                        [ 0.,          0.,          0.04545455,  0.13636364,
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

In [7]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. Sigmoid
2. ReLU
3. Leaky ReLU

Answer:

Sigmoid and ReLu would still have this problem. Sigmoid's gradient would have vanishing gradient for input close to 0 and relu's gradient is either 0 or 1. In one dimensional case, one example sigmoid would have vanishing gradient would be if the input is [-1e5, 1e5] and whereas relu any negative input, i.e [-1, -2, -3]

Leaky ReLU doesn't have this problem because it considers small negative slope when we have negative values, i.e., if x < 0 then alpha else x, where alpha can be equal to 0.01. Hence, no vanishing gradient

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs6353/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [8]:
```
from cs6353.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs6353/layers.py`.

You can make sure that the implementations are correct by running the following:

In [9]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=Fal
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be arc
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error shoul
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs6353/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [10]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=s

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
```

```
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  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]
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

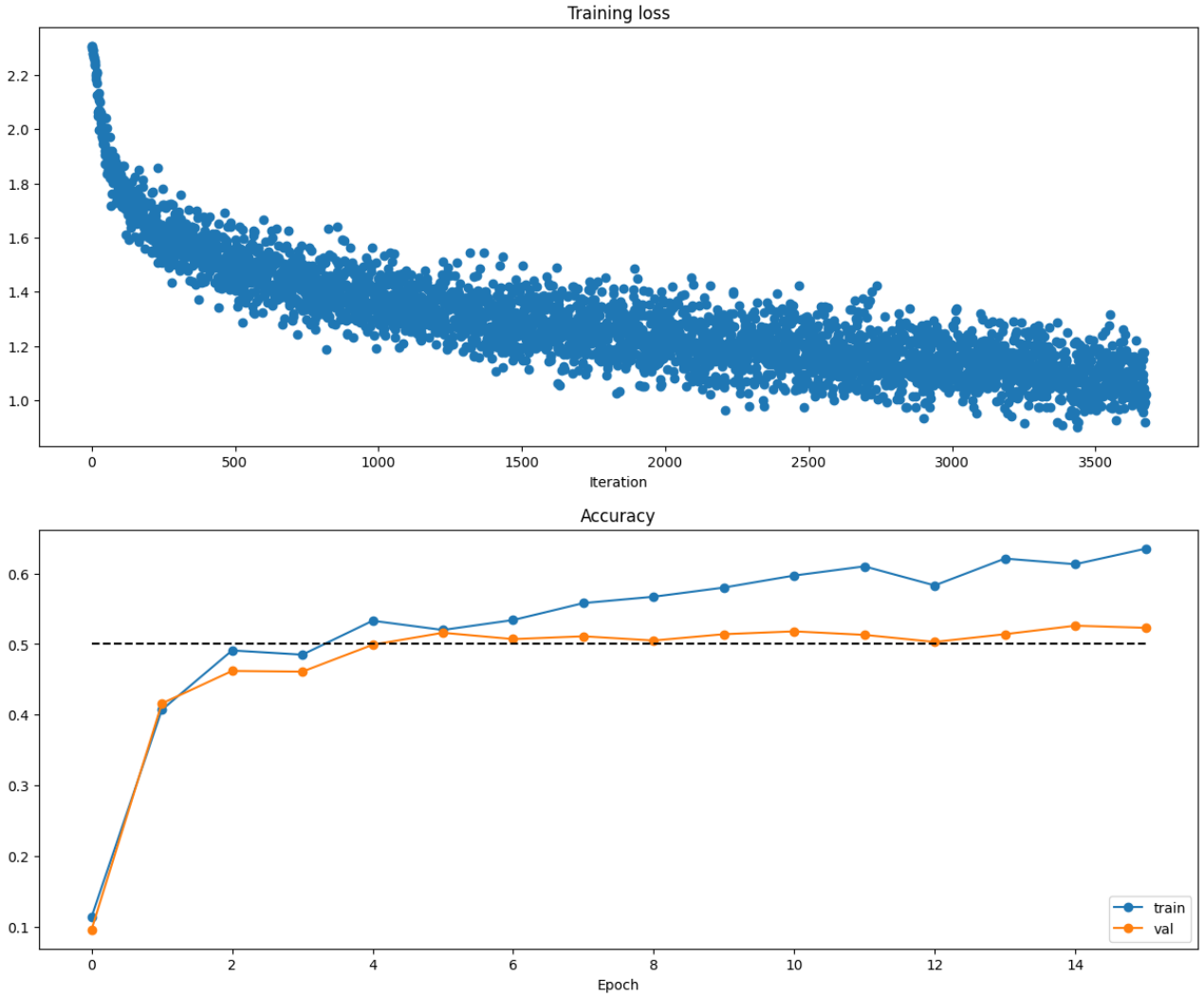
In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Open the file `cs6353/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

In [11]:
```
model = TwoLayerNet()
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least
# 50% accuracy on the validation set.
################################################################################
solver_data = {'X_train': data['X_train'],
               'y_train': data['y_train'],
               'X_val': data['X_val'],
               'y_val': data['y_val']}

# model = TwoLayerNet(input_dim=solver_data['X_train'].shape[1:], hidden_dim
solver = Solver(model, solver_data,
                update_rule='sgd',
                optim_config={'learning_rate': 1e-3,},
                lr_decay = 0.95,
                num_epochs=15,
                batch_size=200,
                print_every=100)

solver.train()
################################################################################
#                                END OF YOUR CODE
################################################################################
```

```
(Iteration 1 / 3675) loss: 2.305965
(Epoch 0 / 15) train acc: 0.114000; val_acc: 0.096000
(Iteration 101 / 3675) loss: 1.856329
(Iteration 201 / 3675) loss: 1.598619
(Epoch 1 / 15) train acc: 0.407000; val_acc: 0.416000
(Iteration 301 / 3675) loss: 1.548536
(Iteration 401 / 3675) loss: 1.661861
(Epoch 2 / 15) train acc: 0.491000; val_acc: 0.462000
(Iteration 501 / 3675) loss: 1.452303
(Iteration 601 / 3675) loss: 1.665413
(Iteration 701 / 3675) loss: 1.295910
(Epoch 3 / 15) train acc: 0.485000; val_acc: 0.461000
(Iteration 801 / 3675) loss: 1.393038
(Iteration 901 / 3675) loss: 1.395341
(Epoch 4 / 15) train acc: 0.533000; val_acc: 0.499000
(Iteration 1001 / 3675) loss: 1.449278
(Iteration 1101 / 3675) loss: 1.276145
(Iteration 1201 / 3675) loss: 1.207275
(Epoch 5 / 15) train acc: 0.520000; val_acc: 0.516000
(Iteration 1301 / 3675) loss: 1.339874
(Iteration 1401 / 3675) loss: 1.430487
(Epoch 6 / 15) train acc: 0.534000; val_acc: 0.507000
(Iteration 1501 / 3675) loss: 1.364778
(Iteration 1601 / 3675) loss: 1.167146
(Iteration 1701 / 3675) loss: 1.362156
(Epoch 7 / 15) train acc: 0.558000; val_acc: 0.511000
(Iteration 1801 / 3675) loss: 1.225039
(Iteration 1901 / 3675) loss: 1.310197
(Epoch 8 / 15) train acc: 0.567000; val_acc: 0.505000
(Iteration 2001 / 3675) loss: 1.271719
(Iteration 2101 / 3675) loss: 1.120800
(Iteration 2201 / 3675) loss: 1.279429
(Epoch 9 / 15) train acc: 0.580000; val_acc: 0.514000
(Iteration 2301 / 3675) loss: 1.229667
(Iteration 2401 / 3675) loss: 1.104043
(Epoch 10 / 15) train acc: 0.597000; val_acc: 0.518000
(Iteration 2501 / 3675) loss: 1.286715
(Iteration 2601 / 3675) loss: 1.143557
(Epoch 11 / 15) train acc: 0.610000; val_acc: 0.513000
(Iteration 2701 / 3675) loss: 1.048077
(Iteration 2801 / 3675) loss: 1.065507
(Iteration 2901 / 3675) loss: 1.130374
(Epoch 12 / 15) train acc: 0.583000; val_acc: 0.503000
(Iteration 3001 / 3675) loss: 1.146657
(Iteration 3101 / 3675) loss: 1.197794
(Epoch 13 / 15) train acc: 0.621000; val_acc: 0.514000
(Iteration 3201 / 3675) loss: 1.045194
(Iteration 3301 / 3675) loss: 0.974399
(Iteration 3401 / 3675) loss: 1.064486
(Epoch 14 / 15) train acc: 0.613000; val_acc: 0.526000
(Iteration 3501 / 3675) loss: 1.043400
(Iteration 3601 / 3675) loss: 1.065562
(Epoch 15 / 15) train acc: 0.635000; val_acc: 0.523000
```

In [12]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs6353/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing batch/layer normalization; we will add those features soon.

Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

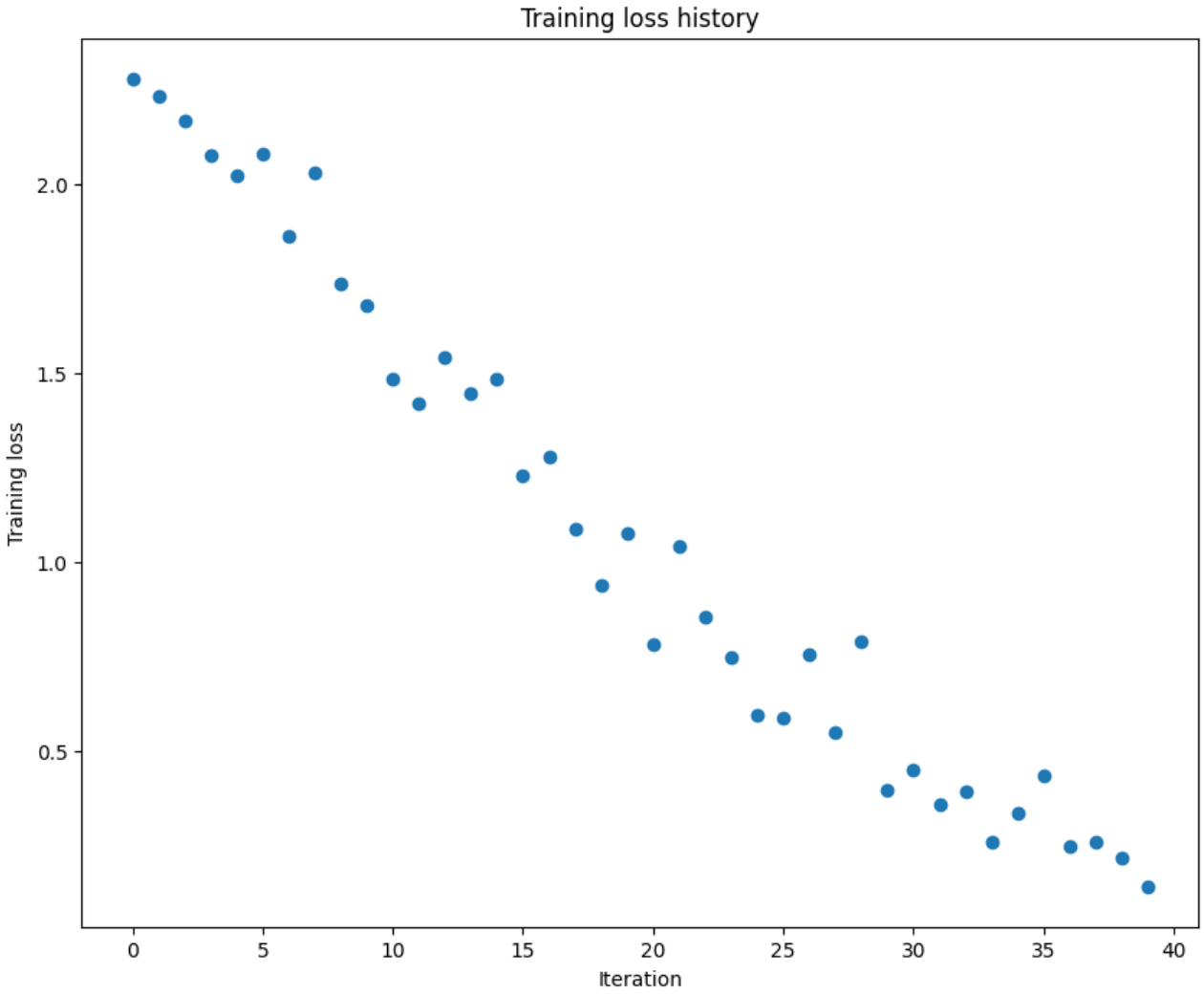
In [13]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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

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

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  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]
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

In [20]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
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 = 1e-2
learning_rate = 5e-3
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

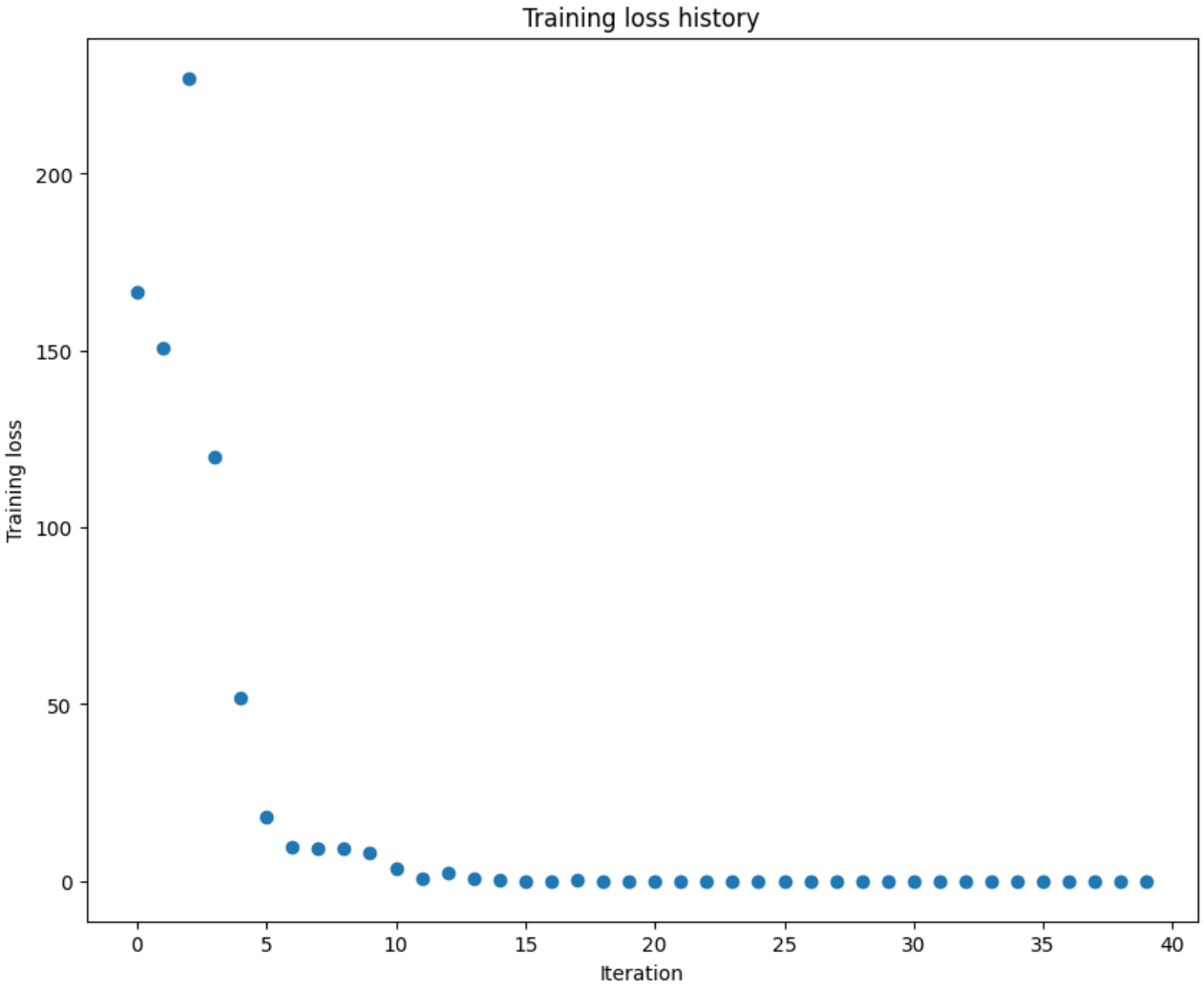
```
(Iteration 1 / 40) loss: 2.280581
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.119000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.154000
(Epoch 2 / 20) train acc: 0.420000; val_acc: 0.146000
(Epoch 3 / 20) train acc: 0.480000; val_acc: 0.133000
(Epoch 4 / 20) train acc: 0.500000; val_acc: 0.173000
(Epoch 5 / 20) train acc: 0.520000; val_acc: 0.182000
(Iteration 11 / 40) loss: 1.486073
(Epoch 6 / 20) train acc: 0.560000; val_acc: 0.183000
(Epoch 7 / 20) train acc: 0.660000; val_acc: 0.199000
(Epoch 8 / 20) train acc: 0.700000; val_acc: 0.175000
(Epoch 9 / 20) train acc: 0.720000; val_acc: 0.170000
(Epoch 10 / 20) train acc: 0.820000; val_acc: 0.199000
(Iteration 21 / 40) loss: 0.782952
(Epoch 11 / 20) train acc: 0.820000; val_acc: 0.178000
(Epoch 12 / 20) train acc: 0.900000; val_acc: 0.203000
(Epoch 13 / 20) train acc: 0.820000; val_acc: 0.182000
(Epoch 14 / 20) train acc: 0.900000; val_acc: 0.197000
(Epoch 15 / 20) train acc: 0.940000; val_acc: 0.202000
(Iteration 31 / 40) loss: 0.449988
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.205000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.216000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.198000
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.192000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.202000
```



Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [15]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
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'],
}

learning_rate = 2e-3
weight_scale = 1e-1
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 166.501707
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.107000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.101000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.122000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.106000
(Epoch 4 / 20) train acc: 0.520000; val_acc: 0.111000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.113000
(Iteration 11 / 40) loss: 3.343141
(Epoch 6 / 20) train acc: 0.840000; val_acc: 0.122000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.113000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.125000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.039138
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.121000
(Iteration 31 / 40) loss: 0.000644
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.121000
```



Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer:

When comparing the difficulty of training these two networks, the five-layer network showed greater sensitivity to the initialization scale. I believe this is because of the increased depth of the network. In deeper networks, setting a small scale can cause the gradients to vanish due to small products, while a large scale can lead to gradient explosion from large products—issues that occur more frequently than in shallower networks. Therefore, finding the appropriate scale for the five-layer network is more challenging compared to the three-layer network, though it can be achieved through random search.

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent.

Open the file `cs6353/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

In [16]:
```
from cs6353.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

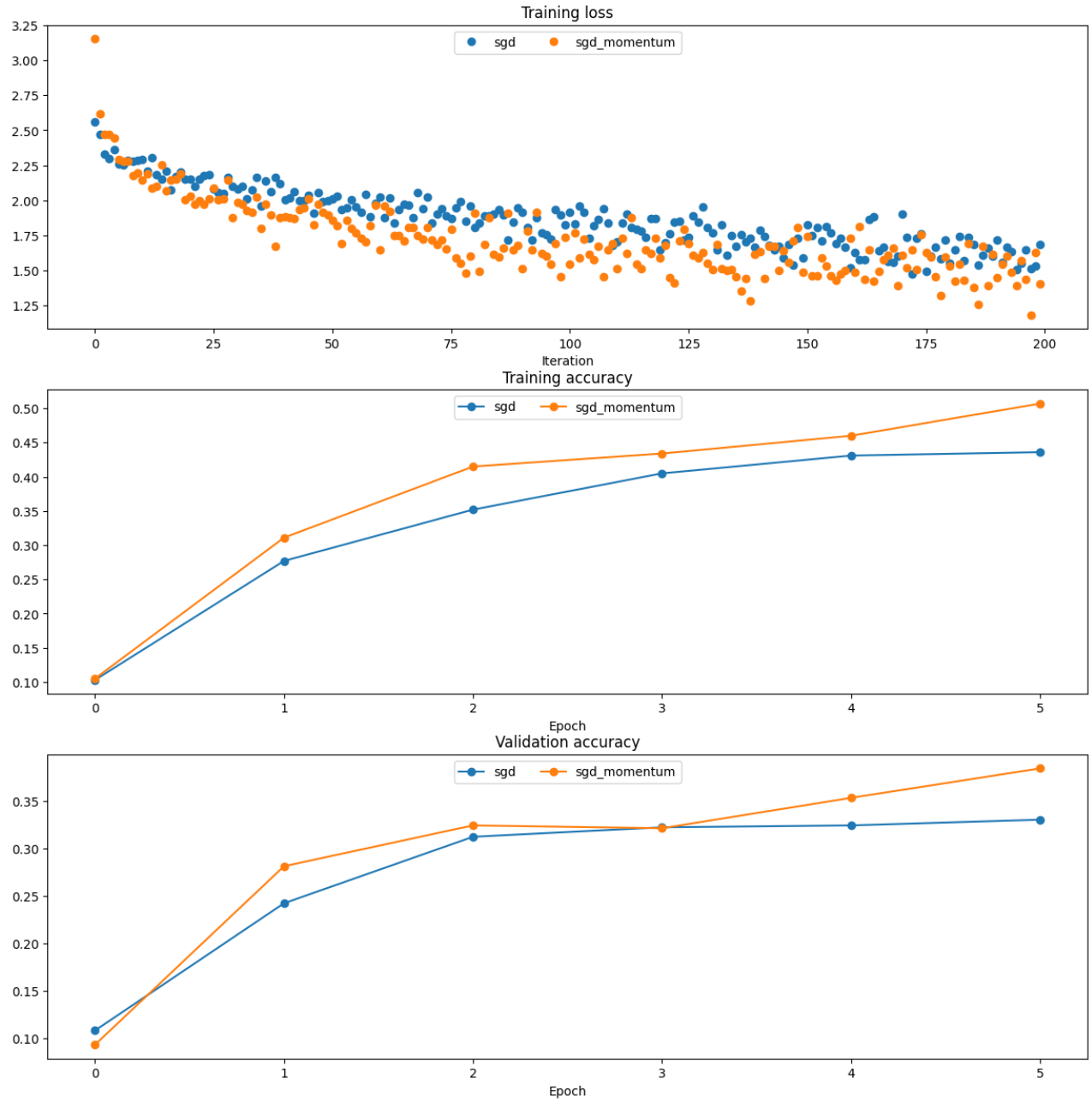
```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

In [17]:
```
num_train = 4000
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'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

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')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

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

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

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

```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082693
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.023753
(Iteration 71 / 200) loss: 2.026621
(Epoch 2 / 5) train acc: 0.352000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.807163
(Iteration 91 / 200) loss: 1.914256
(Iteration 101 / 200) loss: 1.917176
(Iteration 111 / 200) loss: 1.706193
(Epoch 3 / 5) train acc: 0.405000; val_acc: 0.322000
(Iteration 121 / 200) loss: 1.697994
(Iteration 131 / 200) loss: 1.768837
(Iteration 141 / 200) loss: 1.784967
(Iteration 151 / 200) loss: 1.823291
(Epoch 4 / 5) train acc: 0.431000; val_acc: 0.324000
(Iteration 161 / 200) loss: 1.626499
(Iteration 171 / 200) loss: 1.901366
(Iteration 181 / 200) loss: 1.550534
(Iteration 191 / 200) loss: 1.716921
(Epoch 5 / 5) train acc: 0.436000; val_acc: 0.330000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032563
(Iteration 31 / 200) loss: 1.985848
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882353
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512047
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610416
(Iteration 181 / 200) loss: 1.528331
```

```
(Iteration 191 / 200) loss: 1.449159
(Epoch 5 / 5) train acc: 0.507000; val_acc: 0.384000
```



Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` notebook before completing this part, since those techniques can help you train powerful models.

In [65]:
```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might
# find batch/layer normalization useful. Store your best model in  #
# the best_model variable.
################################################################################
from itertools import product
num_train = 50
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'],
}

learning_rates = [0.12894736842105264]
# regularization_strengths = [6.233333333333334e-05]

# learning_rates = np.linspace(1e-2, 1e-1, num=20)
regularization_strengths = np.linspace(6.2e-5, 6.3e-5, num=5)
update_rules = ['sgd', 'sgd_momentum']
weight_scale = [5e-2]
momentum = [0.8, 0.9]

# best validation accuracy: 0.227 with lr 0.12894736842105264, reg 6.2333333

hyperparameters = list(product(*[learning_rates, regularization_strengths, u

best_val = 0
best_lr = 0
best_reg = 0
```

```
best_ur = ''
best_m = 0

for lr, reg, ur, m, w in hyperparameters:
    model = FullyConnectedNet([100, 100, 100, 100, 100],
                    weight_scale=w, reg=reg, dtype=np.float64, normalization
    solver = Solver(model, small_data,
                    print_every=0, num_epochs=50, batch_size=10,
                    update_rule=ur,
                    optim_config={
                    'learning_rate': lr,
                    'momentum': m
                    },
                    verbose=False
             )
    solver.train()

    if solver.best_val_acc > best_val:
        best_val = solver.best_val_acc
        best_lr = lr
        best_reg = reg
        best_ur = ur
        best_m = m
        best_w = w
        best_model = model
        print(f'** New best! best validation accuracy: {best_val} with lr {b


print(f'best validation accuracy: {best_val} with lr {best_lr}, reg {best_re
################################################################################
#                                END OF YOUR CODE
################################################################################
```

```
** New best! best validation accuracy: 0.204 with lr 0.12894736842105264, re
g 6.2e-05, weight scale 0.05 and best update rule sgd and best momentum 0.8
** New best! best validation accuracy: 0.208 with lr 0.12894736842105264, re
g 6.2e-05, weight scale 0.05 and best update rule sgd and best momentum 0.9
** New best! best validation accuracy: 0.214 with lr 0.12894736842105264, re
g 6.225e-05, weight scale 0.05 and best update rule sgd_momentum and best mo
mentum 0.9
** New best! best validation accuracy: 0.217 with lr 0.12894736842105264, re
g 6.275000000000001e-05, weight scale 0.05 and best update rule sgd_momentum
and best momentum 0.9
best validation accuracy: 0.217 with lr 0.12894736842105264, reg 6.275000000
000001e-05, and weight scale 0.05 and best update rule sgd_momentum and best
momentum 0.9
```

best validation accuracy: 0.177 with lr 0.05736152510448681 and weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.204 with lr 0.12689610031679222, reg 1e-05, weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.219 with lr 0.2782559402207124, reg 1e-05, weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.189 with lr 0.1291549665014884, reg 1.2915496650148827e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.189 with lr 0.12, reg 6e-05, and weight scale 0.05 and best update rule sgd_momentum and best momentum 0.9

best validation accuracy: 0.205 with lr 0.12897435897435897, reg 7.000000000000001e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.211 with lr 0.12897435897435897, reg 6.111111111111111e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.22 with lr 0.12894736842105264, reg 6.222222222222222e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.9

best validation accuracy: 0.217 with lr 0.12894736842105264, reg 6e-05, and weight scale 0.05 and best update rule sgd_momentum and best momentum 0.9

best validation accuracy: 0.221 with lr 0.12894736842105264, reg 6.111111111111111e-05, and weight scale 0.05 and best update rule sgd_momentum and best momentum 0.9

best validation accuracy: 0.226 with lr 0.12894736842105264, reg 6e-05, and weight scale 0.05 and best update rule sgd_momentum and best momentum 0.9

best validation accuracy: 0.211 with lr 0.12894736842105264, reg 6e-05, and weight scale 0.05 and best update rule sgd_momentum and best momentum 0.9

best validation accuracy: 0.227 with lr 0.12894736842105264, reg 6.233333333333334e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.9

** New best! best validation accuracy: 0.215 with lr 0.004641588833612777, reg 0.0002154434690031882, weight scale 0.02 and best update rule adam and best momentum 0.99

best validation accuracy: 0.212 with lr 0.12894736842105264, reg 6.2e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.5

** New best! best validation accuracy: 0.226 with lr 0.12894736842105264, reg 6.1e-05, weight scale 0.05 and best update rule sgd and best momentum 0.5

best validation accuracy: 0.216 with lr 0.12894736842105264, reg 6e-05, and weight scale 0.05 and best update rule sgd and best momentum 0.99

Best parameters: {'learning_rate': 0.04814179393818249, 'reg': 6.881218080170343e-05, 'weight_scale': 0.02, 'update_rule': 'adam', 'momentum': 0.8} Best validation accuracy: 0.216

Best parameters: {'learning_rate': 0.04046327613077516, 'reg': 6.322120254343489e-05, 'weight_scale': 0.02, 'update_rule': 'sgd_momentum', 'momentum': 0.9} Best validation accuracy: 0.216

Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

In [66]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.208
Test set accuracy:  0.199
```

In [ ]:
```
```