FullyConnectedNetsContinued

February 15, 2024

[1]:
```python
# 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. 'cse493g1/assignments/assignment3/'
FOLDERNAME = 'cse493g1/assignments/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 COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment3
```

1 Multi-Layer Fully Connected Network Part 2

In this exercise, you will extend your fully connected network from Assignment 2 with Dropout and Normalization Layers. First, you will copy and paste all the necessary parts from Assignment 2. Then you will re-train your model from A2 as a baseline. Next, you will complete the batchnorm and dropout notebook, and then return to this notebook and create an improved model using dropout and normalization.

[2]:
```python
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
```

```
from cse493g1.classifiers.fc_net import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.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))))
```

[3]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Copy necessary parts from A2.

Fill in the following functions by copying and pasting your answers from A2: `affine_forward` in `cse493g1/layers.py` `affine_backward` in `cse493g1/layers.py` `relu_forward` in `cse493g1/layers.py` `relu_backward` in `cse493g1/layers.py` `softmax_loss` in `cse493g1/layers.py` `sgd_momentum` in `cse493g1/optim.py` `rmsprop` in `cse493g1/optim.py` `adam` in `cse493g1/optim.py`

3 Train baseline model from A2

Copy and Paste your `FullyConnectedNet` model from `cse493g1/classifiers/fc_net.py` in Assignment 2 into `FullyConnectedNetBasic` in the file `cse493g1/classifiers/fc_net.py` in this assignment. Use the best hyperparms that you found from the previous assignment to train this model. Call this model `best_model_basic`

```
[ ]: best_model_basic = None

     ################################################################################
     # TODO: Train the best FullyConnectedNetBasic that you can on CIFAR-10. Store␣
      ↪your best model in  #
     # the best_model_basic variable.                                              ␣
      ↪     #
     ################################################################################
     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

     solvers = {}
     num_train = 10000
     new_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 = [1e-3, 5e-4]
     update_rules = ['rmsprop']
     weight_scale = [1e-2, 5e-2]
     best_acc = -1

     for update_rule in update_rules:
       for lr in learning_rates:
         for weight in weight_scale:
           model = FullyConnectedNet(
                   [100, 100, 100, 100, 100],
                   weight_scale=weight
               )

           solver = Solver(
               model,
               data,
               num_epochs=5,
               batch_size=100,
               update_rule=update_rule,
               optim_config={'learning_rate': lr},
               verbose=True
           )
           solvers[update_rule] = solver
           solver.train()

           if solver.best_val_acc > best_acc:
             best_acc = solver.best_val_acc
             best_model_basic = model
```

```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

4 Evaluate baseline model from A2

Evaluate above baseline model.

[6]:
```
y_test_pred = np.argmax(best_model_basic.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model_basic.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:  0.505
Test set accuracy:  0.507
```

5 Train improved model

Design a new model in `FullyConnectedNetImproved` in the file `cse493g1/classifiers/fc_net.py`. You can start by having `FullyConnectedNetImproved` be the same design as `FullyConnectedNetBasic`. Next, complete the BatchNormoralization.ipynb and Dropout.ipynb notebooks. Then return to this notebook and complete `FullyConnectedNetImproved` by adding in batchnorm and dropout. Try to beat the accuracy of your baseline model! You may have to adjust your hyperparameters.

[ ]:
```
best_model_improved = None

################################################################################
# TODO: Train the best FullyConnectedNetImproved that you can on CIFAR-10. You␣
 ↪might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_mode_improved variable.                                             ␣
 ↪       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

solvers = {}
num_train = 10000
new_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 = [1e-3, 5e-4]
update_rules = ['adam']
```

```
weight_scale = [5e-2]
best_acc = -1

for update_rule in update_rules:
  for lr in learning_rates:
    for weight in weight_scale:
      model = FullyConnectedNetImproved(
              [100, 100, 100, 100, 100],
              dropout_keep_ratio=0.75,
              normalization='batchnorm',
              weight_scale=weight
          )

      solver = Solver(
          model,
          data,
          num_epochs=25,
          batch_size=100,
          update_rule=update_rule,
          optim_config={'learning_rate': lr},
          verbose=True
      )
      solvers[update_rule] = solver
      solver.train()

      if solver.best_val_acc > best_acc:
        best_acc = solver.best_val_acc
        best_model_improved = model

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

6 Test Your Model!

Run your best model on the validation and test sets. Are you able to outperform the baseline model that has no Batchnorm or Dropout?

[12]:
```
y_test_pred = np.argmax(best_model_improved.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model_improved.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:  0.553
Test set accuracy:  0.563
```

BatchNormalization

February 15, 2024

[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. 'cse493g1/assignments/assignment3/'
FOLDERNAME = 'cse493g1/assignments/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 COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment3
```

1 Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization, proposed by [1] in 2015.

To understand the goal of batch normalization, it is important to first recognize that machine learning methods tend to perform better with input data consisting 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, they propose to insert into the network layers that normalize batches. At training time, such a layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

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

[2]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cse493g1.classifiers.fc_net import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.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(f"  means: {x.mean(axis=axis)}")
    print(f"  stds:  {x.std(axis=axis)}\n")
```

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
```

```
assignment.
        The instructions to do this will be given in a section of the notebook
below.
```

[107]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Batch Normalization: Forward Pass

In the file `cse493g1/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

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

[3]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network.
np.random.seed(493)
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: [-10.78452322  16.26772621 -30.25276687]
  stds:  [29.12943373 37.33627641 30.56256822]

After batch normalization (gamma=1, beta=0)
  means: [ 9.32587341e-17  1.06581410e-16 -1.70419234e-16]
  stds:  [0.99999999 1.         0.99999999]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means: [11. 12. 13.]
  stds:  [0.99999999 1.99999999 2.99999998]
```

[4]:
```
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.

np.random.seed(493)
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: [ 0.02116281 -0.00291808  0.05612367]
stds:  [1.10717728 1.01804534 0.99964292]
```

3 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.

[5]:
```
# Gradient check batchnorm backward pass.
np.random.seed(493)
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:  9.471264828334034e-10
dgamma error:  7.346136578328727e-11
dbeta error:  8.753834990721456e-12
```

4 Batch Normalization: Alternative Backward Pass

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

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

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

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

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

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

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

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

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

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

After doing so, implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

[6]:
```
np.random.seed(493)
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:  2.0394489922050502e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 2.05x
```

5 Fully Connected Networks with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNetImproved` in the file `cse493g1/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 `cse493g1/layer_utils.py`.

[141]:
```
np.random.seed(493)
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 = FullyConnectedNetImproved([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64,
                            normalization='batchnorm')

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

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
  ↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.076971674776514
W3 relative error: 2.86e-10
b3 relative error: 9.58e-11

Running check with reg =  3.14
Initial loss:  3.4311320680719737
W3 relative error: 2.46e-07
b3 relative error: 1.66e-10
```

6 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.

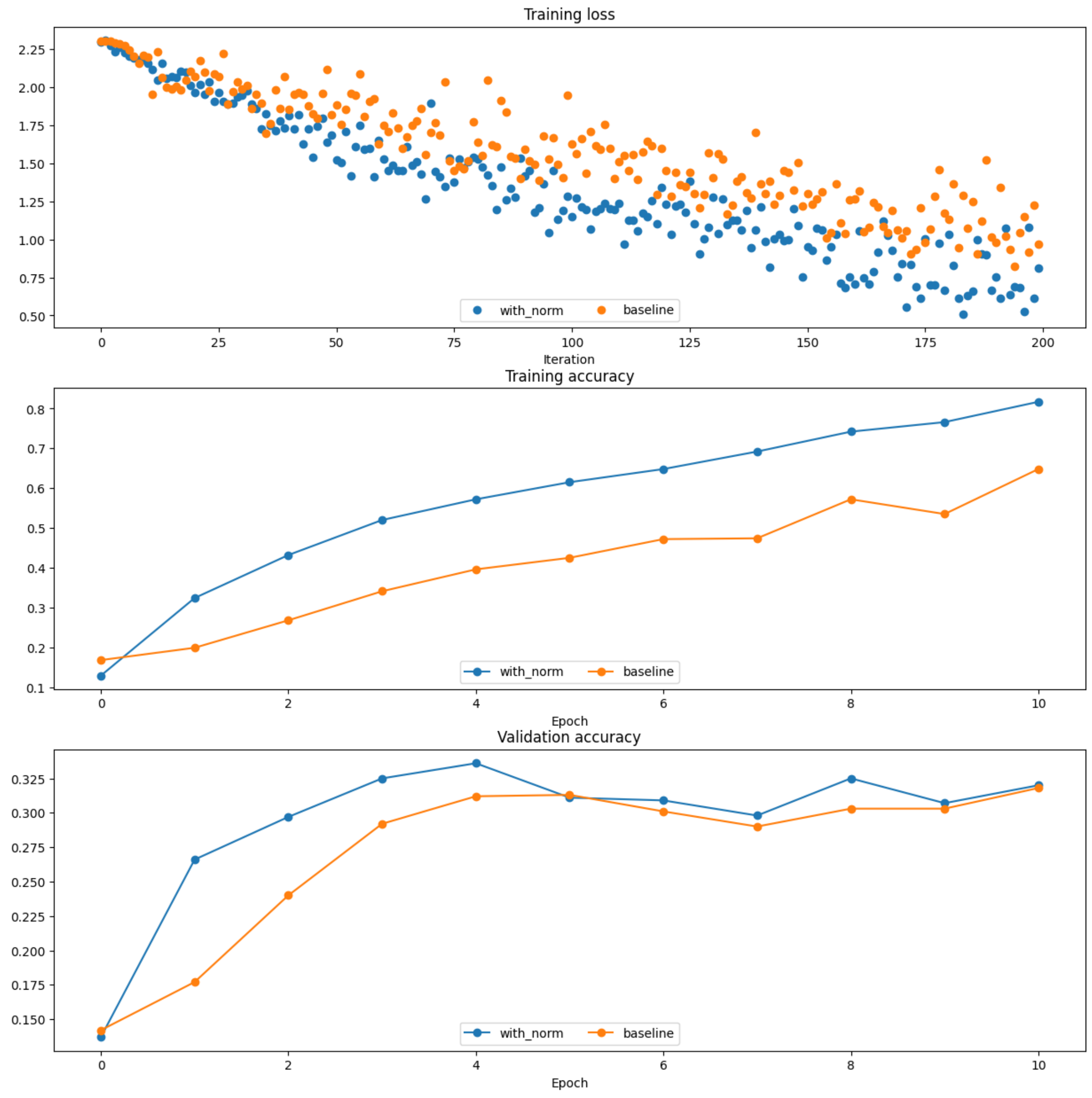
[145]:
```
np.random.seed(493)

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

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

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

```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.296650
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.137000
(Epoch 1 / 10) train acc: 0.324000; val_acc: 0.266000
(Iteration 21 / 200) loss: 1.966126
(Epoch 2 / 10) train acc: 0.432000; val_acc: 0.297000
(Iteration 41 / 200) loss: 1.812367
(Epoch 3 / 10) train acc: 0.520000; val_acc: 0.325000
(Iteration 61 / 200) loss: 1.527469
(Epoch 4 / 10) train acc: 0.572000; val_acc: 0.336000
(Iteration 81 / 200) loss: 1.526125
(Epoch 5 / 10) train acc: 0.615000; val_acc: 0.311000
(Iteration 101 / 200) loss: 1.148105
(Epoch 6 / 10) train acc: 0.648000; val_acc: 0.309000
(Iteration 121 / 200) loss: 1.231244
(Epoch 7 / 10) train acc: 0.692000; val_acc: 0.298000
(Iteration 141 / 200) loss: 1.214927
(Epoch 8 / 10) train acc: 0.742000; val_acc: 0.325000
(Iteration 161 / 200) loss: 0.706635
(Epoch 9 / 10) train acc: 0.766000; val_acc: 0.307000
(Iteration 181 / 200) loss: 1.035320
(Epoch 10 / 10) train acc: 0.817000; val_acc: 0.320000

Solver without batch norm:
(Iteration 1 / 200) loss: 2.303307
(Epoch 0 / 10) train acc: 0.168000; val_acc: 0.142000
(Epoch 1 / 10) train acc: 0.199000; val_acc: 0.177000
(Iteration 21 / 200) loss: 2.072841
(Epoch 2 / 10) train acc: 0.268000; val_acc: 0.240000
(Iteration 41 / 200) loss: 1.855935
(Epoch 3 / 10) train acc: 0.341000; val_acc: 0.292000
(Iteration 61 / 200) loss: 1.747505
(Epoch 4 / 10) train acc: 0.396000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.641977
```

```
(Epoch 5 / 10) train acc: 0.425000; val_acc: 0.313000
(Iteration 101 / 200) loss: 1.626354
(Epoch 6 / 10) train acc: 0.472000; val_acc: 0.301000
(Iteration 121 / 200) loss: 1.450363
(Epoch 7 / 10) train acc: 0.474000; val_acc: 0.290000
(Iteration 141 / 200) loss: 1.363606
(Epoch 8 / 10) train acc: 0.572000; val_acc: 0.303000
(Iteration 161 / 200) loss: 1.266012
(Epoch 9 / 10) train acc: 0.535000; val_acc: 0.303000
(Iteration 181 / 200) loss: 1.133536
(Epoch 10 / 10) train acc: 0.648000; val_acc: 0.318000
```

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.

[146]:
```python
def plot_training_history(title, label, baseline, bn_solvers, plot_fn,␣
 ↪bl_marker='.', bn_marker='.', labels=None):
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.train_acc_history, bl_marker='-o',␣
 ↪bn_marker='-o')
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.val_acc_history, bl_marker='-o',␣
 ↪bn_marker='-o')

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



7 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 eight-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.

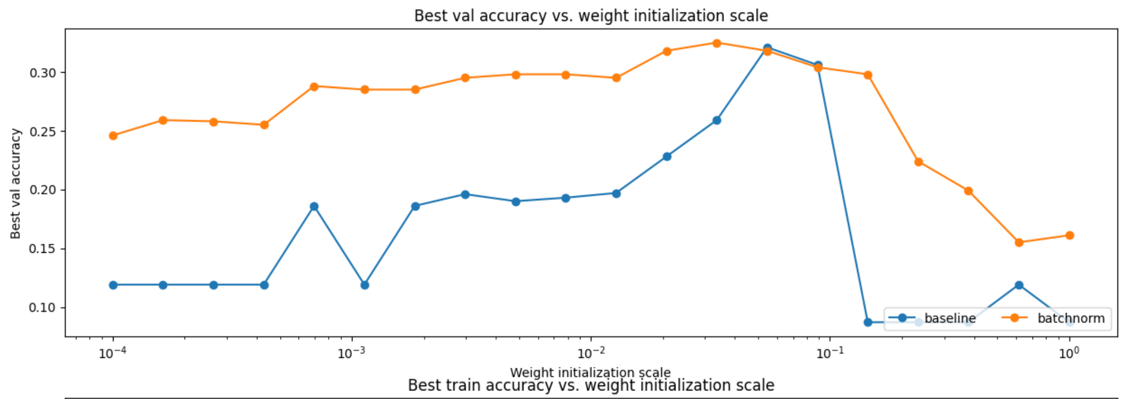
[147]:
```
np.random.seed(493)

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

    bn_solver = Solver(bn_model, small_data,
                    num_epochs=10, batch_size=50,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    verbose=False, print_every=200)
    bn_solver.train()
    bn_solvers_ws[weight_scale] = bn_solver

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

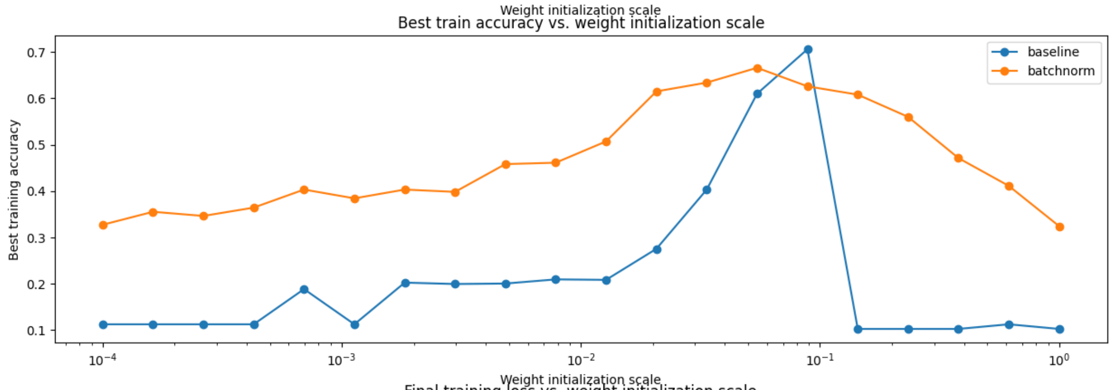
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
```

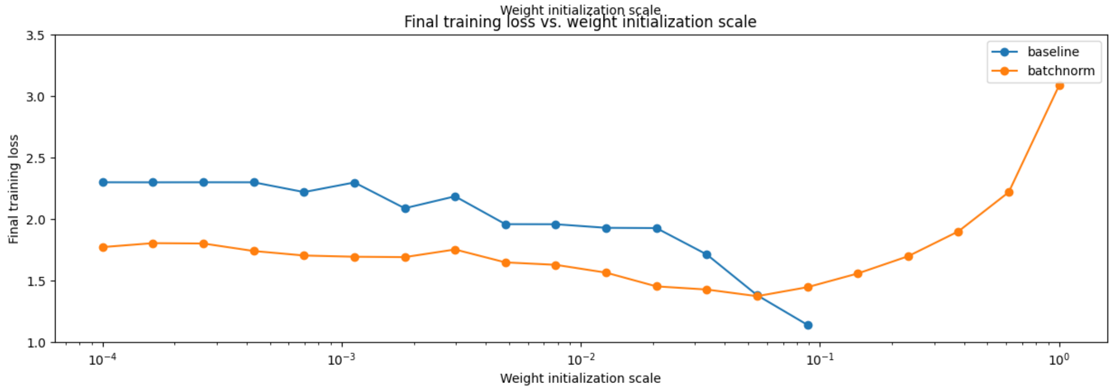
```
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/layers.py:200:
RuntimeWarning: overflow encountered in exp
  s_j_exp = np.sum(np.exp(x), axis=1) # N
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/layers.py:201:
RuntimeWarning: overflow encountered in exp
  softmax_probs = np.exp(x) / s_j_exp[:, np.newaxis] # NxC
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/layers.py:201:
RuntimeWarning: invalid value encountered in divide
  softmax_probs = np.exp(x) / s_j_exp[:, np.newaxis] # NxC
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/layers.py:203:
RuntimeWarning: divide by zero encountered in log
  loss = np.sum(-np.log(correct_class_scores)) / num_train # scalar
```

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

[148]:
```
# 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()
```







7.1 Inline Question 1:

Describe the results of this experiment. How does the weight initialization scale affect models with/without batch normalization differently, and why?

7.2 Answer:

- Overall models with batch normalization(bn) out perform models without bn. Models with bn. usually have higher training and val accuracy, and lower training loss.
- Models with bn are more tolerant with weight initialization scale change. Even when the weight initialization scale is bad, the models with bn still performs okay comparing to models without bn.
- Because batch normalization adjusts the mean and variance of each layer's activations, it

makes the model less sensitive to the scale of weight initialization. Bn makes deep neural networks much easier to train. It improves gradient flow, and helps avoid the vanishing/exploding gradient problem.

8 Batch Normalization and Batch Size

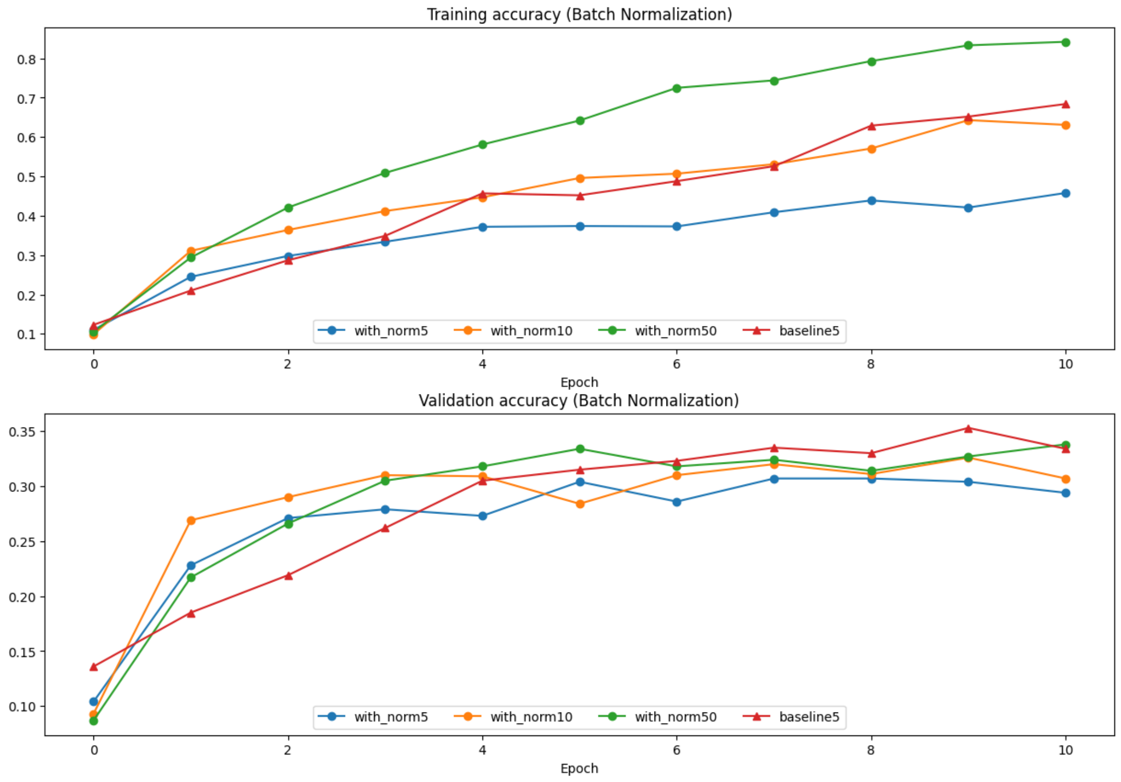
We will now run a small experiment to study the interaction of batch normalization and batch size.

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.

[149]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(493)

    # 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 = FullyConnectedNetImproved(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': lr,
                    },
                    verbose=False)
    solver.train()

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

    return bn_solvers, solver, batch_sizes

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

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

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

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



8.1 Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

8.2 Answer:

- The model with batch normalization and a batch size of 50 shows the highest training accuracy, followed by batch sizes of 10 and 5. The training accuracy increases with the size of the batch when batch normalization is used.
- The model with batch normalization and a batch size of 50 also has the highest validation accuracy, though the models with batch sizes of 10 and 5 are not far behind. The validation accuracy for models with batch normalization plateaus after a certain number of epochs, with little difference observed between batch sizes of 10 and 50 towards the end of the epochs.
- Larger batch sizes with batch normalization tend to yield better training performance. This is likely because larger batches provide a more accurate estimation of the population mean and variance, which batch normalization uses to scale and shift the activations.
- The relationship is observed becasue: Batch normalization benefits from larger batch sizes because the mean and variance are estimated more accurately with more samples, leading to a more stable learning process and faster convergence during training. However, larger batch sizes may also introduce the issue of overfitting on training data.

9 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.

9.1 Inline Question 3:

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

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

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

9.2 Answer:

- Batch Normalization: 3
- Layer Normalization: 1, 2

10 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 `cse493g1/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. * In `cse493g1/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cse493g1/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNetImproved`. 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.

[158]:
```python
# 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(493)
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: [-13.55016261  10.48803421   0.51090326  16.1646569 ]
  stds:  [10.43227939 44.64705887 61.08455449 28.07932638]

After layer normalization (gamma=1, beta=0)
  means: [ 1.85037171e-17 -3.70074342e-17  0.00000000e+00  7.40148683e-17]
  stds:  [0.99999995 1.         1.         0.99999999]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means: [5. 5. 5. 5.]
  stds:  [2.99999986 2.99999999 3.         2.99999998]
```

[175]:
```python
# Gradient check batchnorm backward pass.
np.random.seed(493)
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))
```

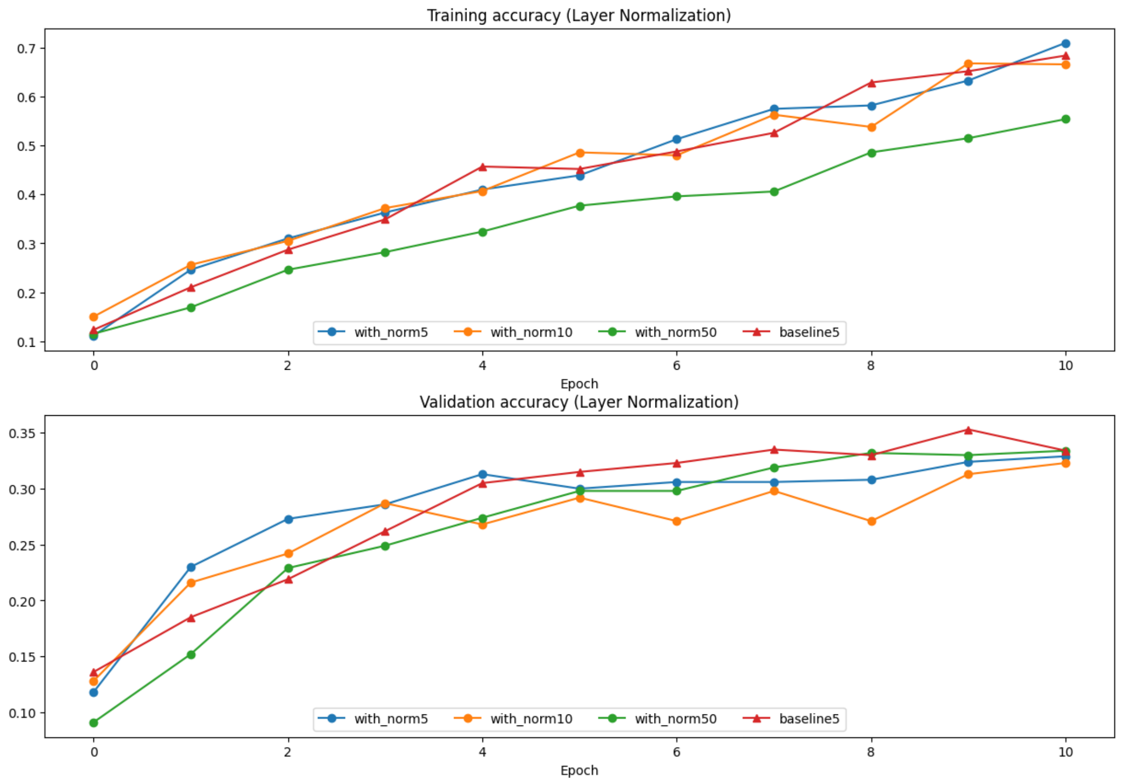
```
dx error:  7.425248660880946e-10
dgamma error:  1.3843700526756892e-11
dbeta error:  8.753834990721456e-12
```

11 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!

[176]:
```
ln_solvers_bsize, solver_bsize, batch_sizes =␣
↪run_batchsize_experiments('layernorm')

plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Layer Normalization)','Epoch',␣
↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Layer Normalization)','Epoch',␣
↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
↪bn_marker='-o', labels=batch_sizes)

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

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



11.1 Inline Question 4:

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

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

11.2 Answer:

2 is not likely to work well. Because if the number of features is very small, the mean and variance estimates may not be very robust, since they are based on a small sample size.

Dropout

February 15, 2024

[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. 'cse493g1/assignments/assignment3/'
FOLDERNAME = 'cse493g1/assignments/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 COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment3
```

1 Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some output activations to zero during the forward pass. In this exercise, you will implement a dropout layer and modify your fully connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

[2]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
```

```
from cse493g1.classifiers.fc_net import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.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))))
```

```
=========== You can safely ignore the message below if you are NOT working on
ConvolutionalNetworks.ipynb ===========
        You will need to compile a Cython extension for a portion of this
assignment.
        The instructions to do this will be given in a section of the notebook
below.
```

[3]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Dropout: Forward Pass

In the file `cse493g1/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

[12]:
```
np.random.seed(493)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
    out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
    out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

    print('Running tests with p = ', p)
    print('Mean of input: ', x.mean())
    print('Mean of train-time output: ', out.mean())
    print('Mean of test-time output: ', out_test.mean())
    print('Fraction of train-time output set to zero: ', (out == 0).mean())
    print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
    print()
```

```
Running tests with p =  0.25
Mean of input:  9.998754292315606
Mean of train-time output:  10.013822700086228
Mean of test-time output:  9.998754292315606
Fraction of train-time output set to zero:  0.749592
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  9.998754292315606
Mean of train-time output:  9.961259703367766
Mean of test-time output:  9.998754292315606
Fraction of train-time output set to zero:  0.601452
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  9.998754292315606
Mean of train-time output:  9.983016036268745
Mean of test-time output:  9.998754292315606
Fraction of train-time output set to zero:  0.3011
Fraction of test-time output set to zero:  0.0
```

3 Dropout: Backward Pass

In the file `cse493g1/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[14]:
```
np.random.seed(493)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,␣
 ↪dropout_param)[0], x, dout)
```

```
# Error should be around e-10 or less.
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.445599254826896e-11
```

3.1 Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2 Answer:

The expected sum of the outputs from the dropout layer will be different at training time compared to test time, which can significantly affect the performance of the neural network. That is becasuetThe size of output is reduced since we reduced the size of input by the mask, and we did nothing to compensate.

4 Fully Connected Networks with Dropout

In the file `cse493g1/classifiers/fc_net.py`, copy your `FullyConnectedNetBasic` into `FullyConnectedNetImproved` and modify this new net to use dropout. Specifically, if the constructor of the network receives a value that is not 1 for the `dropout_keep_ratio` parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

[16]:
```
np.random.seed(493)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout_keep_ratio in [1, 0.75, 0.5]:
    print('Running check with dropout = ', dropout_keep_ratio)
    model = FullyConnectedNetImproved(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        weight_scale=5e-2,
        dtype=np.float64,
        dropout_keep_ratio=dropout_keep_ratio,
        seed=123
    )

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

    # Relative errors should be around e-6 or less.
    # Note that it's fine if for dropout_keep_ratio=1 you have W2 error be on␣
↪the order of e-5.
    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name],␣
↪verbose=False, h=1e-5)
        print('%s relative error: %.2e' % (name, rel_error(grad_num,␣
↪grads[name])))
    print()
```

```
Running check with dropout =  1
Initial loss:  2.299821914918452
W1 relative error: 9.66e-08
W2 relative error: 4.43e-06
W3 relative error: 1.41e-06
b1 relative error: 9.10e-09
b2 relative error: 1.36e-07
b3 relative error: 1.34e-10

Running check with dropout =  0.75
Initial loss:  2.303014228386485
W1 relative error: 2.04e-07
W2 relative error: 8.49e-08
W3 relative error: 2.07e-07
b1 relative error: 3.90e-08
b2 relative error: 2.71e-09
b3 relative error: 1.00e-10

Running check with dropout =  0.5
Initial loss:  2.30486036957328
W1 relative error: 6.66e-08
W2 relative error: 1.09e-08
W3 relative error: 3.47e-08
b1 relative error: 2.08e-09
b2 relative error: 8.34e-10
b3 relative error: 8.54e-11
```

5 Regularization Experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

[19]:
```
# Train two identical nets, one with dropout and one without.
np.random.seed(493)
num_train = 500
```

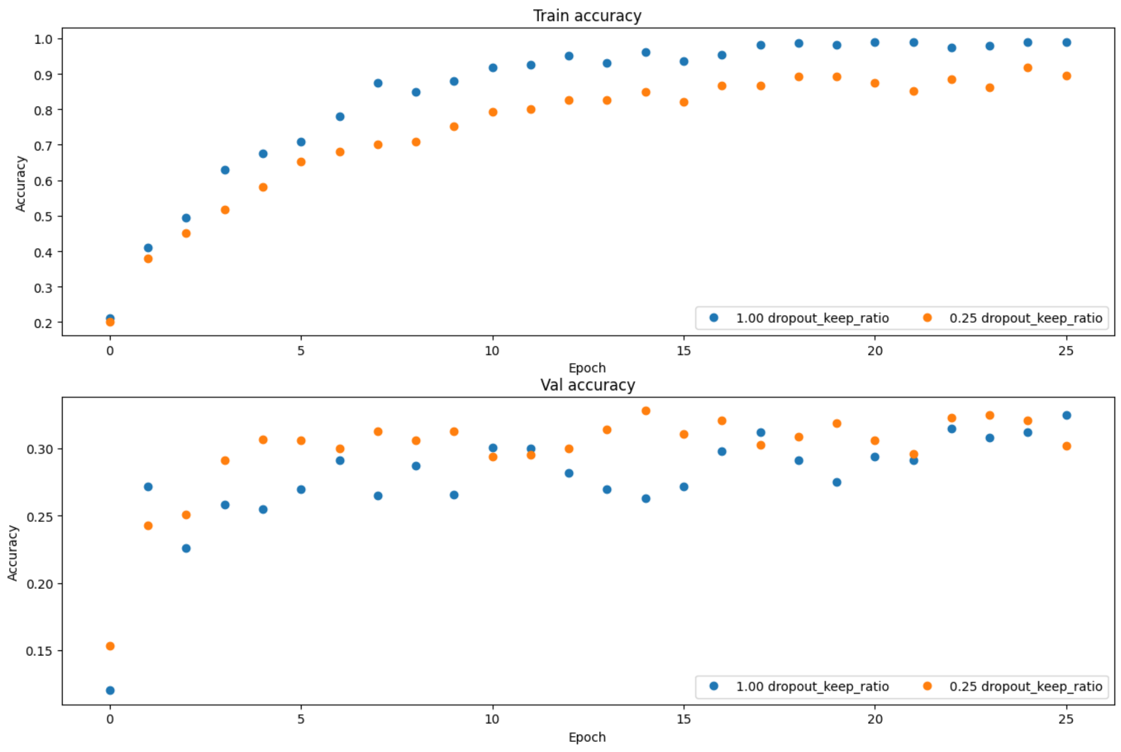
```
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout_keep_ratio in dropout_choices:
    model = FullyConnectedNetImproved(
        [500],
        dropout_keep_ratio=dropout_keep_ratio
    )
    print(dropout_keep_ratio)

    solver = Solver(
        model,
        small_data,
        num_epochs=25,
        batch_size=100,
        update_rule='adam',
        optim_config={'learning_rate': 5e-4,},
        verbose=True,
        print_every=100
    )
    solver.train()
    solvers[dropout_keep_ratio] = solver
    print()
```

```
1
(Iteration 1 / 125) loss: 7.287606
(Epoch 0 / 25) train acc: 0.212000; val_acc: 0.120000
(Epoch 1 / 25) train acc: 0.410000; val_acc: 0.272000
(Epoch 2 / 25) train acc: 0.494000; val_acc: 0.226000
(Epoch 3 / 25) train acc: 0.630000; val_acc: 0.258000
(Epoch 4 / 25) train acc: 0.676000; val_acc: 0.255000
(Epoch 5 / 25) train acc: 0.710000; val_acc: 0.270000
(Epoch 6 / 25) train acc: 0.780000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.874000; val_acc: 0.265000
(Epoch 8 / 25) train acc: 0.848000; val_acc: 0.287000
(Epoch 9 / 25) train acc: 0.880000; val_acc: 0.266000
(Epoch 10 / 25) train acc: 0.918000; val_acc: 0.301000
(Epoch 11 / 25) train acc: 0.926000; val_acc: 0.300000
(Epoch 12 / 25) train acc: 0.950000; val_acc: 0.282000
(Epoch 13 / 25) train acc: 0.930000; val_acc: 0.270000
(Epoch 14 / 25) train acc: 0.962000; val_acc: 0.263000
```

```
(Epoch 15 / 25) train acc: 0.936000; val_acc: 0.272000
(Epoch 16 / 25) train acc: 0.954000; val_acc: 0.298000
(Epoch 17 / 25) train acc: 0.982000; val_acc: 0.312000
(Epoch 18 / 25) train acc: 0.986000; val_acc: 0.291000
(Epoch 19 / 25) train acc: 0.982000; val_acc: 0.275000
(Epoch 20 / 25) train acc: 0.990000; val_acc: 0.294000
(Iteration 101 / 125) loss: 0.007025
(Epoch 21 / 25) train acc: 0.990000; val_acc: 0.291000
(Epoch 22 / 25) train acc: 0.974000; val_acc: 0.315000
(Epoch 23 / 25) train acc: 0.980000; val_acc: 0.308000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.312000
(Epoch 25 / 25) train acc: 0.990000; val_acc: 0.325000

0.25
(Iteration 1 / 125) loss: 16.330096
(Epoch 0 / 25) train acc: 0.202000; val_acc: 0.153000
(Epoch 1 / 25) train acc: 0.380000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.452000; val_acc: 0.251000
(Epoch 3 / 25) train acc: 0.518000; val_acc: 0.291000
(Epoch 4 / 25) train acc: 0.580000; val_acc: 0.307000
(Epoch 5 / 25) train acc: 0.652000; val_acc: 0.306000
(Epoch 6 / 25) train acc: 0.682000; val_acc: 0.300000
(Epoch 7 / 25) train acc: 0.702000; val_acc: 0.313000
(Epoch 8 / 25) train acc: 0.708000; val_acc: 0.306000
(Epoch 9 / 25) train acc: 0.752000; val_acc: 0.313000
(Epoch 10 / 25) train acc: 0.792000; val_acc: 0.294000
(Epoch 11 / 25) train acc: 0.800000; val_acc: 0.295000
(Epoch 12 / 25) train acc: 0.826000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.826000; val_acc: 0.314000
(Epoch 14 / 25) train acc: 0.850000; val_acc: 0.328000
(Epoch 15 / 25) train acc: 0.820000; val_acc: 0.311000
(Epoch 16 / 25) train acc: 0.866000; val_acc: 0.321000
(Epoch 17 / 25) train acc: 0.868000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.892000; val_acc: 0.309000
(Epoch 19 / 25) train acc: 0.892000; val_acc: 0.319000
(Epoch 20 / 25) train acc: 0.874000; val_acc: 0.306000
(Iteration 101 / 125) loss: 5.687084
(Epoch 21 / 25) train acc: 0.852000; val_acc: 0.296000
(Epoch 22 / 25) train acc: 0.886000; val_acc: 0.323000
(Epoch 23 / 25) train acc: 0.862000; val_acc: 0.325000
(Epoch 24 / 25) train acc: 0.918000; val_acc: 0.321000
(Epoch 25 / 25) train acc: 0.896000; val_acc: 0.302000
```

[20]:
```
# Plot train and validation accuracies of the two models.
train_accs = []
val_accs = []
```

```python
for dropout_keep_ratio in dropout_choices:
    solver = solvers[dropout_keep_ratio]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].train_acc_history, 'o', label='%.2f␣
 ↪dropout_keep_ratio' % dropout_keep_ratio)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].val_acc_history, 'o', label='%.2f␣
 ↪dropout_keep_ratio' % dropout_keep_ratio)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

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



5.1 Inline Question 2:

Compare the validation and training accuracies with and without dropout – what do your results suggest about dropout as a regularizer?

5.2 Answer:

The model without dropout have higher training accuarcy and lower val accuracy than model with dropout. This suggests the model without dropout is overfitting on training data, and dropout can improve the performance of the model as a good regularizer.

ConvolutionalNetworks

February 15, 2024

[ ]:
```
# 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. 'cse493g1/assignments/assignment3/'
FOLDERNAME = 'cse493g1/assignments/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 COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignments/assignment3/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignments/assignment3
```

1 Convolutional Networks

So far we have worked with deep fully connected networks, using them to explore different optimization strategies and network architectures. Fully connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

[ ]:
```
# Setup cell.
import numpy as np
import matplotlib.pyplot as plt
from cse493g1.classifiers.cnn import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient_array,␣
↪eval_numerical_gradient
from cse493g1.layers import *
from cse493g1.fast_layers import *
from cse493g1.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

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

[ ]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

2 Convolution: Naive Forward Pass

The core of a convolutional network is the convolution operation. In the file `cse493g1/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

[ ]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
```

```
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                           [-0.18387192, -0.2109216 ]],
                          [[ 0.21027089,  0.21661097],
                           [ 0.22847626,  0.23004637]],
                          [[ 0.50813986,  0.54309974],
                           [ 0.64082444,  0.67101435]]],
                         [[[-0.98053589, -1.03143541],
                           [-1.19128892, -1.24695841]],
                          [[ 0.69108355,  0.66880383],
                           [ 0.59480972,  0.56776003]],
                          [[ 2.36270298,  2.36904306],
                           [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

2.1 Aside: Image Processing via Convolutions

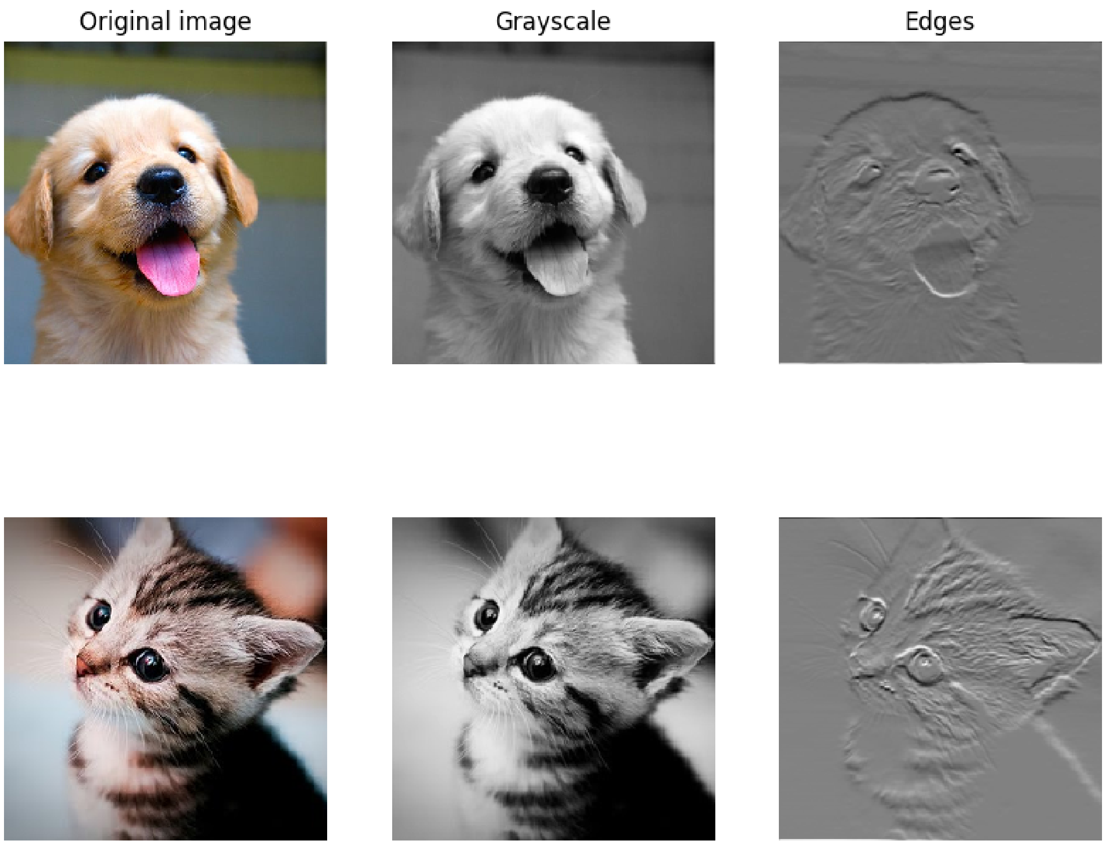
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

[ ]:
```
from imageio import imread
from PIL import Image

kitten = imread('cse493g1/notebook_images/kitten.jpg')
puppy = imread('cse493g1/notebook_images/puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size)))
x = np.zeros((2, 3, img_size, img_size))
```

```
x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_no_ax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
```

```
imshow_no_ax(out[1, 1])
plt.show()
```

```
<ipython-input-6-ed6e0fc1d3d2>:4: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  kitten = imread('cse493g1/notebook_images/kitten.jpg')
<ipython-input-6-ed6e0fc1d3d2>:5: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  puppy = imread('cse493g1/notebook_images/puppy.jpg')
```



3 Convolution: Naive Backward Pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cse493g1/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

```
[ ]: np.random.seed(493)
     x = np.random.randn(4, 3, 5, 5)
     w = np.random.randn(2, 3, 3, 3)
     b = np.random.randn(2,)
     dout = np.random.randn(4, 2, 5, 5)
     conv_param = {'stride': 1, 'pad': 1}

     dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b,␣
      ↪conv_param)[0], x, dout)
     dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,␣
      ↪conv_param)[0], w, dout)
     db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,␣
      ↪conv_param)[0], b, dout)

     out, cache = conv_forward_naive(x, w, b, conv_param)
     dx, dw, db = conv_backward_naive(dout, cache)

     # Your errors should be around e-8 or less.
     print('Testing conv_backward_naive function')
     print('dx error: ', rel_error(dx, dx_num))
     print('dw error: ', rel_error(dw, dw_num))
     print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  3.001996916775403e-09
dw error:  3.4403330681980457e-10
db error:  4.314875022006533e-10
```

4 Max-Pooling: Naive Forward Pass

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cse493g1/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
[ ]: x_shape = (2, 3, 4, 4)
     x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
     pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

     out, _ = max_pool_forward_naive(x, pool_param)

     correct_out = np.array([[[[-0.26315789, -0.24842105],
                               [-0.20421053, -0.18947368]],
                              [[-0.14526316, -0.13052632],
                               [-0.08631579, -0.07157895]],
                                   [[-0.02736842, -0.01263158],
                                    [ 0.03157895,  0.04631579]]],
                                  [[[ 0.09052632,  0.10526316],
                                    [ 0.14947368,  0.16421053]],
                                   [[ 0.20842105,  0.22315789],
                                    [ 0.26736842,  0.28210526]],
                                   [[ 0.32631579,  0.34105263],
                                    [ 0.38526316,  0.4       ]]]])

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

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

5 Max-Pooling: Naive Backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cse493g1/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

[ ]:
```
np.random.seed(493)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x,␣
 ↪pool_param)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

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

```
Testing max_pool_backward_naive function:
dx error:  3.2756208694894018e-12
```

6 Fast Layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cse493g1/fast_layers.py`.

6.0.1 Execute the below cell, save the notebook, and restart the runtime

The fast convolution implementation depends on a Cython extension; to compile it, run the cell below. Next, save the Colab notebook (`File > Save`) and **restart the runtime** (`Runtime > Restart runtime`). You can then re-execute the preceeding cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

```
[ ]: # Remember to restart the runtime after executing this cell!
     %cd /content/drive/My\ Drive/$FOLDERNAME/cse493g1/
     !python setup.py build_ext --inplace
     %cd /content/drive/My\ Drive/$FOLDERNAME/
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

Note: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

```
[ ]: # Rel errors should be around e-9 or less.
     from cse493g1.fast_layers import conv_forward_fast, conv_backward_fast
     from time import time
     np.random.seed(493)
     x = np.random.randn(100, 3, 31, 31)
     w = np.random.randn(25, 3, 3, 3)
     b = np.random.randn(25,)
     dout = np.random.randn(100, 25, 16, 16)
     conv_param = {'stride': 2, 'pad': 1}

     t0 = time()
     out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
     t1 = time()
     out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
     t2 = time()

     print('Testing conv_forward_fast:')
     print('Naive: %fs' % (t1 - t0))
     print('Fast: %fs' % (t2 - t1))
     print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
     print('Difference: ', rel_error(out_naive, out_fast))

     t0 = time()
     dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
     t1 = time()
```

```
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 0.173399s
Fast: 0.008874s
Speedup: 19.540799x
Difference:  6.9816349604656895e-12

Testing conv_backward_fast:
Naive: 0.465094s
Fast: 0.006563s
Speedup: 70.861455x
dx difference:  4.1285209892297247e-11
dw difference:  8.57970676643518e-14
db difference:  0.0
```

[ ]:
```
# Relative errors should be close to 0.0.
from cse493g1.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(493)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
```

```
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.007358s
fast: 0.003208s
speedup: 2.293230x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.016985s
fast: 0.010269s
speedup: 1.653936x
dx difference:  0.0
```

7 Convolutional “Sandwich” Layers

In the previous assignment, we introduced the concept of “sandwich” layers that combine multiple operations into commonly used patterns. In the file `cse493g1/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check their usage.

[ ]:
```
from cse493g1.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(493)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w,␣
  ↪b, conv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w,␣
  ↪b, conv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w,␣
  ↪b, conv_param, pool_param)[0], b, dout)
```

```
# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
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 conv_relu_pool
dx error:  3.0063520770488714e-08
dw error:  6.194086751504152e-10
db error:  3.2819548156748916e-07
```

[ ]:
```
from cse493g1.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(493)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
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 conv_relu:
dx error:  2.556535683363290e-09
dw error:  1.5835432103810795e-09
db error:  1.926243366624518e-10
```

8 Three-Layer Convolutional Network

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cse493g1/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

8.1 Sanity Check Loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization the loss should go up slightly.

```
[ ]: model = ThreeLayerConvNet()

     N = 50
     X = np.random.randn(N, 3, 32, 32)
     y = np.random.randint(10, size=N)

     loss, grads = model.loss(X, y)
     print('Initial loss (no regularization): ', loss)

     model.reg = 0.5
     loss, grads = model.loss(X, y)
     print('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.3025875115011694
Initial loss (with regularization):  2.508816587290621
```

8.2 Gradient Check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

```
[ ]: num_inputs = 2
     input_dim = (3, 16, 16)
     reg = 0.0
     num_classes = 10
     np.random.seed(473)
     X = np.random.randn(num_inputs, *input_dim)
     y = np.random.randint(num_classes, size=num_inputs)

     model = ThreeLayerConvNet(
         num_filters=3,
         filter_size=3,
         input_dim=input_dim,
         hidden_dim=7,
         dtype=np.float64
     )
     loss, grads = model.loss(X, y)
     # Errors should be small, but correct implementations may have
     # relative errors up to the order of e-2
     for param_name in sorted(grads):
         f = lambda _: model.loss(X, y)[0]
```

```
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[param_name])))
```

```
W1 max relative error: 2.455028e-04
W2 max relative error: 1.886091e-03
W3 max relative error: 6.430690e-06
b1 max relative error: 1.672084e-05
b2 max relative error: 2.339602e-08
b3 max relative error: 1.659283e-09
```

8.3 Overfit Small Data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

[ ]:
```
np.random.seed(493)

num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(
    model,
    small_data,
    num_epochs=15,
    batch_size=50,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=1
)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.377935
(Epoch 0 / 15) train acc: 0.140000; val_acc: 0.110000
(Iteration 2 / 30) loss: 3.507474
(Epoch 1 / 15) train acc: 0.170000; val_acc: 0.119000
(Iteration 3 / 30) loss: 4.381055
```

```
(Iteration 4 / 30) loss: 3.571722
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.097000
(Iteration 5 / 30) loss: 2.401453
(Iteration 6 / 30) loss: 2.270795
(Epoch 3 / 15) train acc: 0.230000; val_acc: 0.098000
(Iteration 7 / 30) loss: 2.451352
(Iteration 8 / 30) loss: 2.472043
(Epoch 4 / 15) train acc: 0.290000; val_acc: 0.156000
(Iteration 9 / 30) loss: 2.028178
(Iteration 10 / 30) loss: 1.889899
(Epoch 5 / 15) train acc: 0.450000; val_acc: 0.182000
(Iteration 11 / 30) loss: 1.845204
(Iteration 12 / 30) loss: 1.650750
(Epoch 6 / 15) train acc: 0.310000; val_acc: 0.164000
(Iteration 13 / 30) loss: 1.722978
(Iteration 14 / 30) loss: 1.817188
(Epoch 7 / 15) train acc: 0.500000; val_acc: 0.200000
(Iteration 15 / 30) loss: 1.494970
(Iteration 16 / 30) loss: 1.549230
(Epoch 8 / 15) train acc: 0.580000; val_acc: 0.178000
(Iteration 17 / 30) loss: 1.428538
(Iteration 18 / 30) loss: 1.449242
(Epoch 9 / 15) train acc: 0.620000; val_acc: 0.180000
(Iteration 19 / 30) loss: 1.337344
(Iteration 20 / 30) loss: 1.045957
(Epoch 10 / 15) train acc: 0.610000; val_acc: 0.182000
(Iteration 21 / 30) loss: 1.052689
(Iteration 22 / 30) loss: 1.299899
(Epoch 11 / 15) train acc: 0.670000; val_acc: 0.185000
(Iteration 23 / 30) loss: 1.076239
(Iteration 24 / 30) loss: 0.885635
(Epoch 12 / 15) train acc: 0.750000; val_acc: 0.203000
(Iteration 25 / 30) loss: 0.777443
(Iteration 26 / 30) loss: 0.860241
(Epoch 13 / 15) train acc: 0.820000; val_acc: 0.212000
(Iteration 27 / 30) loss: 0.725782
(Iteration 28 / 30) loss: 0.645265
(Epoch 14 / 15) train acc: 0.870000; val_acc: 0.202000
(Iteration 29 / 30) loss: 0.480345
(Iteration 30 / 30) loss: 0.449025
(Epoch 15 / 15) train acc: 0.890000; val_acc: 0.197000
```

[ ]:
```python
# Print final training accuracy.
print(
    "Small data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

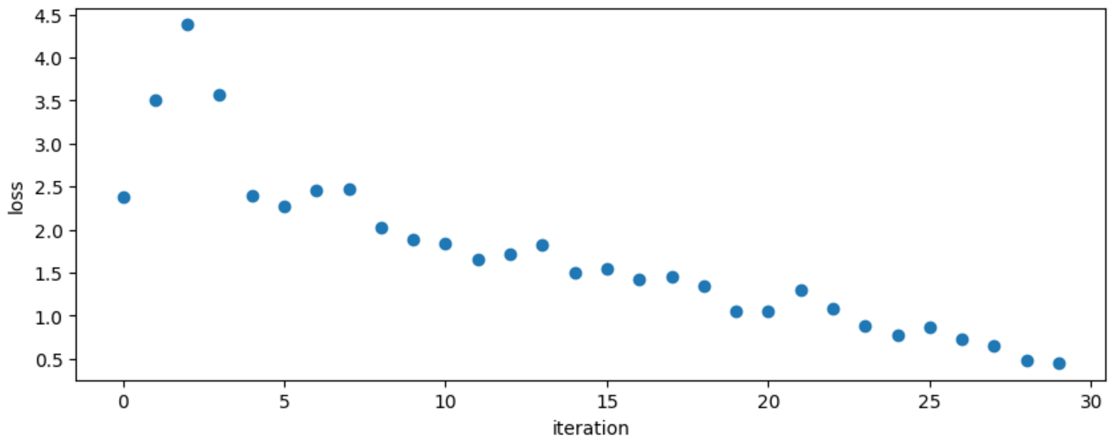
```
Small data training accuracy: 0.82
```

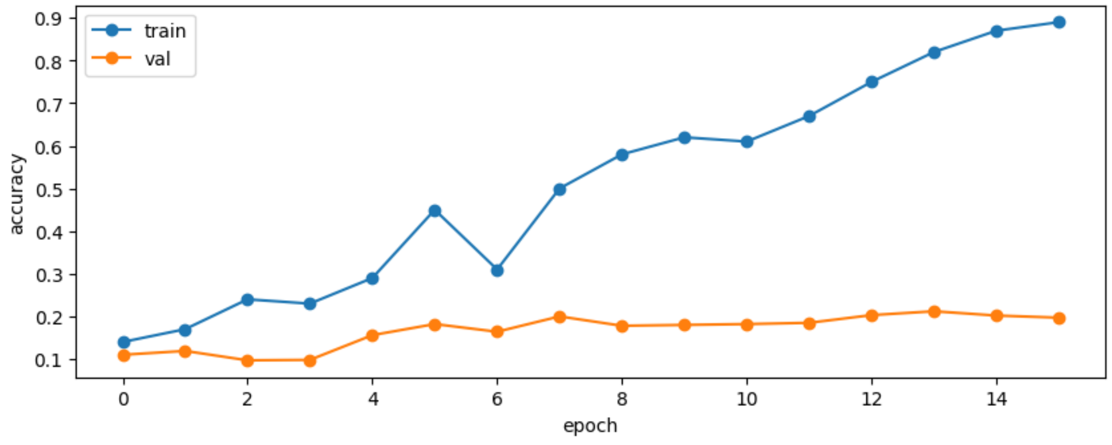
[ ]:
```
# Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

```
Small data validation accuracy: 0.212
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

[ ]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```





8.4 Train the Network

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

[ ]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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

```
(Iteration 1 / 980) loss: 2.304620
(Epoch 0 / 1) train acc: 0.100000; val_acc: 0.104000
(Iteration 21 / 980) loss: 2.133366
(Iteration 41 / 980) loss: 2.210642
(Iteration 61 / 980) loss: 1.698347
(Iteration 81 / 980) loss: 2.180361
(Iteration 101 / 980) loss: 1.833051
(Iteration 121 / 980) loss: 2.064002
(Iteration 141 / 980) loss: 1.889457
(Iteration 161 / 980) loss: 1.794145
(Iteration 181 / 980) loss: 1.818827
(Iteration 201 / 980) loss: 1.938266
(Iteration 221 / 980) loss: 1.629249
(Iteration 241 / 980) loss: 1.739791
(Iteration 261 / 980) loss: 1.860865
(Iteration 281 / 980) loss: 1.911879
(Iteration 301 / 980) loss: 1.773437
(Iteration 321 / 980) loss: 1.812807
(Iteration 341 / 980) loss: 1.554410
(Iteration 361 / 980) loss: 1.358729
(Iteration 381 / 980) loss: 1.790137
(Iteration 401 / 980) loss: 1.907026
(Iteration 421 / 980) loss: 1.957498
(Iteration 441 / 980) loss: 1.289243
(Iteration 461 / 980) loss: 1.904827
(Iteration 481 / 980) loss: 1.568783
(Iteration 501 / 980) loss: 1.478035
(Iteration 521 / 980) loss: 1.389411
(Iteration 541 / 980) loss: 1.525054
(Iteration 561 / 980) loss: 1.445444
(Iteration 581 / 980) loss: 1.601799
(Iteration 601 / 980) loss: 1.921848
(Iteration 621 / 980) loss: 1.470561
(Iteration 641 / 980) loss: 2.109136
(Iteration 661 / 980) loss: 1.305093
(Iteration 681 / 980) loss: 1.765340
(Iteration 701 / 980) loss: 1.591652
(Iteration 721 / 980) loss: 1.633411
(Iteration 741 / 980) loss: 1.609258
(Iteration 761 / 980) loss: 1.392624
(Iteration 781 / 980) loss: 1.853354
(Iteration 801 / 980) loss: 1.576398
(Iteration 821 / 980) loss: 1.296810
(Iteration 841 / 980) loss: 1.436183
(Iteration 861 / 980) loss: 1.788521
(Iteration 881 / 980) loss: 1.639770
(Iteration 901 / 980) loss: 1.407738
(Iteration 921 / 980) loss: 1.841744
```

```
(Iteration 941 / 980) loss: 1.590494
(Iteration 961 / 980) loss: 1.354923
(Epoch 1 / 1) train acc: 0.490000; val_acc: 0.499000
```

[ ]:
```
# Print final training accuracy.
print(
    "Full data training accuracy:",
    solver.check_accuracy(data['X_train'], data['y_train'])
)
```

```
Full data training accuracy: 0.4910204081632653
```

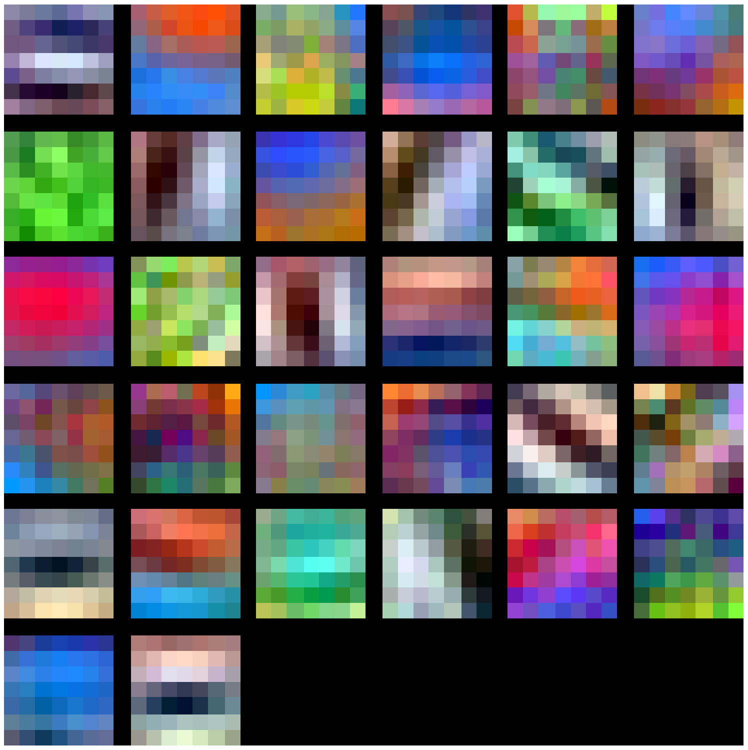
[ ]:
```
# Print final validation accuracy.
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
Full data validation accuracy: 0.499
```

8.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

[ ]:
```
from cse493g1.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



9 EXTRA CREDIT: Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully connected networks. As proposed in the original paper (link in `BatchNormalization.ipynb`), batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally, batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect every feature channel's statistics e.g. mean, variance to be relatively consistent both between different images, and different locations within the same image – after all, every feature channel is produced by the same convolutional filter! Therefore, spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over the minibatch dimension `N` as well the spatial dimensions `H` and `W`.

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

10 EXTRA CREDIT: Spatial Batch Normalization: Forward Pass

In the file `cse493g1/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

```
[ ]: np.random.seed(493)

     # Check the training-time forward pass by checking means and variances
     # of features both before and after spatial batch normalization.
     N, C, H, W = 2, 3, 4, 5
     x = 4 * np.random.randn(N, C, H, W) + 10

     print('Before spatial batch normalization:')
     print('  shape: ', x.shape)
     print('  means: ', x.mean(axis=(0, 2, 3)))
     print('  stds: ', x.std(axis=(0, 2, 3)))

     # Means should be close to zero and stds close to one
     gamma, beta = np.ones(C), np.zeros(C)
     bn_param = {'mode': 'train'}
     out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
     print('After spatial batch normalization:')
     print('  shape: ', out.shape)
     print('  means: ', out.mean(axis=(0, 2, 3)))
     print('  stds: ', out.std(axis=(0, 2, 3)))

     # Means should be close to beta and stds close to gamma
     gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
     out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
     print('After spatial batch normalization (nontrivial gamma, beta):')
     print('  shape: ', out.shape)
     print('  means: ', out.mean(axis=(0, 2, 3)))
     print('  stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  shape:  (2, 3, 4, 5)
  means:  [9.2630585  9.90177254 9.94773619]
  stds:  [3.57082624 3.61057967 4.35995843]
After spatial batch normalization:
  shape:  (2, 3, 4, 5)
  means:  [-1.28785871e-15 -7.10542736e-16 -1.22124533e-16]
  stds:  [0.99999961 0.99999962 0.99999974]
After spatial batch normalization (nontrivial gamma, beta):
  shape:  (2, 3, 4, 5)
  means:  [6. 7. 8.]
  stds:  [2.99999882 3.99999847 4.99999868]
```

[ ]:
```
np.random.seed(493)

# 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.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, 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 spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [ 0.04646792  0.0557893   0.02508963 -0.01599525]
  stds:  [0.97801211 0.99502029 1.0071852  1.00599827]
```

11 EXTRA CREDIT: Spatial Batch Normalization: Backward Pass

In the file `cse493g1/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[ ]:
```
np.random.seed(493)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
```

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

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

12 EXTRA CREDIT: Spatial Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [2] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [3] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups and a per-group per-datapoint normalization instead.

Visual comparison of the normalization techniques discussed so far (image edited from [3])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional computer vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [4] – after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization.

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

[3] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

13 EXTRA CREDIT: Spatial Group Normalization: Forward Pass

In the file `cse493g1/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[ ]:
```
np.random.seed(493)

# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization.
N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  shape: ', x.shape)
print('  means: ', x_g.mean(axis=1))
print('  stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  shape: ', out.shape)
print('  means: ', out_g.mean(axis=1))
print('  stds: ', out_g.std(axis=1))
```

14 EXTRA CREDIT: Spatial Group Normalization: Backward Pass

In the file `cse493g1/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[ ]:
```
np.random.seed(493)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
```

```python
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

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

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)

# You should expect errors of magnitudes between 1e-12 and 1e-07.
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```