hw5

Yuanteng Chen

3039125444 1. Depthwise Separable Convolutions

(a). learnable parameters: $(3\times3\times3)\times4=108$

(b)

$$\begin{cases} \text{Depthwise convolution}: & 3\times3\times3=27 \\ \text{Pointwise convolution}: & (1\times1\times3)\times4=12 \end{cases}$$

learnable parameters: $27+12=39$

2. Regularization and dropout

$$L(w)=\|y-Xw\|_2^2 \quad (1)$$

$$L(\tilde{w})=E_{R\sim\text{Bernoulli}(p)}\left[\|y-(R\odot X)\tilde{w}\|_2^2\right] \quad (2)$$

$$L(w)=\|y-Xw\|_2^2+\|\Gamma w\|_2^2 \quad (3)$$

(a) manipulate (2) to eliminate the expectations and get. $L(\tilde{w})=\|y-pX\tilde{w}\|_2^2+p(1-p)\|\tilde{\Gamma}\tilde{w}\|_2^2$

Solution:

assume $R\odot X=P$

$$\therefore \|y-(R\odot X)\tilde{w}\|_2^2=\|y-P\tilde{w}\|_2^2$$

$$=y^Ty+w^TP^TPw-2w^TP^Ty$$

$\therefore E_{R\sim \text{Bernoulli}(p)}[\|y-(R\odot X)\tilde{w}\|_2^2]$

$= E_{R\sim B(p)}[y^Ty + w^Tp^Tpw - 2w^Tp^Ty]$

① $E_R[p]_{ij} = E_R[(R\odot X)_{ij}] = E_R[R_{ij}]\cdot X_{ij} = pX_{ij}$

② $E_R[2w^Tp^Ty] = 2pw^TX^Ty$

③ $(E_R[(p^Tp)])_{ij} = \sum_{k=1}^N E_R[R_{ki}R_{kj}X_{ki}X_{kj}]$

$$E_R[(p^Tp)]_{ij} = \begin{cases} \sum_{k=1}^N E_R(R_{ki})E_R(R_{kj})\cdot X_{ki}\cdot X_{kj} = p^2(X^TX)_{ij} & (i\neq j) \\ \sum_{k=1}^N E_R[R_{ki}^2 X_{ki}X_{kj}] = \sum_{k=1}^N E_R[R_{ki}^2]X_{ki}X_{kj} = p(X^TX)_{ij} & (i=j) \end{cases}$$

$$(E_R[(p^Tp)])_{ij} - p^2(X^TX)_{ij} = \begin{cases} 0 & i\neq j \\ (p^2-p)(X^TX)_{ij} & i=j \end{cases}$$

$\therefore E_{R\sim B(p)}[y^Ty + w^Tp^Tpw - 2w^Tp^Ty]$

$= y^Ty - 2pw^TX^Ty + (p^2w^TX^TXw - p^2w^TX^TXw) + w^TE_R[p^Tp]w$

$= (y^Ty - 2pw^TX^Ty + p^2w^TX^TXw) - p^2w^TX^TXw + w^TE_R[p^Tp]w$

$= \|y-pXw\|_2^2 + w^T(E_R[p^Tp] - p^2X^TX)w$

$= \|y-pXw\|_2^2 + w^T(p^2-p)\,\text{diag}(X^TX)w$

(only when $i=j$. $(E_R[(p^Tp)])_{ij} - p^2(X^TX)_{ij} = (p^2-p)(X^TX)_{ij}$)

$= \|y-pXw\|_2^2 + p(1-p)\|\tilde{\Gamma}w\|_2^2 \quad \tilde{\Gamma} = \sqrt{\text{diag}(X^TX)}$

(b). $L(\tilde{w}) = \|y - pX\tilde{w}\|_2^2 + p(1-p)\|\tilde{\Gamma}\tilde{w}\|_2^2$

assume $w = p\tilde{w}$

$L(\tilde{w}) = \|y - Xw\|_2^2 + p(1-p)\|\tilde{\Gamma}\frac{w}{p}\|_2^2$

$= \|y - Xw\|_2^2 + \|\sqrt{\frac{1-p}{p}}\tilde{\Gamma}w\|_2^2$

$= \|y - Xw\|_2^2 + \|\Gamma w\|_2^2 \quad (\Gamma = \sqrt{\frac{1-p}{p}}\tilde{\Gamma})$

(c) $L(w) = \|y - Xw\|_2^2 + \|\Gamma w\|_2^2$

$\downarrow$

$L(\tilde{w}) = \|y - \tilde{X}\tilde{w}\|_2^2 + \lambda\|\tilde{w}\|_2^2$

Sol: assume $\tilde{w} = \Gamma w \quad \therefore w = \Gamma^{-1}\tilde{w}$

$\therefore \tilde{X}\Gamma = X \quad \tilde{X} = X \cdot \Gamma^{-1}$

3. Multiplicative Regularization beyond Dropout

expected training loss:

$L(w) = E_{R_{ij} \sim N(\mu, \sigma^2)}[\|y - (R \odot X)w\|_2^2]$

can be put in the form:

$L(w) = \|y - \_(A)\_ Xw\|_2^2 + \_(B)\_\|\Gamma w\|_2^2$

where $\Gamma = (\text{diag}(X^TX))^{\frac{1}{2}}$

Sol:

in 2(a).

$E_{R \sim B(p)}[\|y - (R \odot X)w\|_2^2]$

$= \|y - pXw\|_2^2 + p(1-p)\|\tilde{\Gamma}w\|_2^2 \quad \tilde{\Gamma} = (\text{diag}(X^TX))^{\frac{1}{2}}$

in Bernoulli distribution :

$P(X=k) = p^k (1-p)^{1-k}$

$E(X) = p$, $Var(X) = p(1-p)$

in normal distribution

$E(X) = \mu$. $Var(X) = \sigma^2$

A: μ B: σ^2

4. Analyzing Distributed Training

	Number of Message sent	Size of each message
All-to-all	$n(n-1)$	p
Parameter Server	$2n$	p
Ring All-Reduce	$n(2(n-1))$	$\frac{p}{n}$

Setup Environment

If you are working on this assignment using Google Colab, please execute the codes below.

Alternatively, you can also do this assignment using a local anaconda environment (or a Python virtualenv). Please clone the GitHub repo by running `git clone https://github.com/Berkeley-CS182/cs182hw3.git` and refer to `README.md` for further details.

In [ ]:
```
#@title Mount your Google Drive

import os
from google.colab import drive
drive.mount('/content/gdrive')
```

In [ ]:
```
#@title Set up mount symlink

DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw3_sp23'
DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '')
if not os.path.exists(DRIVE_PYTHON_PATH):
  %mkdir $DRIVE_PATH

## the space in `My Drive` causes some issues,
## make a symlink to avoid this
SYM_PATH = '/content/cs182hw3'
if not os.path.exists(SYM_PATH):
  !ln -s $DRIVE_PATH $SYM_PATH
```

In [ ]:
```
#@title Install dependencies

!pip install numpy==1.21.6 imageio==2.9.0 matplotlib==3.2.2
```

In [ ]:
```
#@title Clone homework repo

%cd $SYM_PATH
if not os.path.exists("cs182hw3"):
  !git clone https://github.com/Berkeley-CS182/cs182hw3.git
%cd cs182hw3
```

In [4]:
```
#@title Download datasets

%cd deeplearning/datasets/
!bash ./get_datasets.sh
%cd ../..

#instead we load the cifar-10 dataset from local file
```

```
f:\new_gitee_code\berkeley_class\Deep_Learning\hw5\code\q_coding_bn_drop_cnn\deep
learning\datasets
f:\new_gitee_code\berkeley_class\Deep_Learning\hw5\code\q_coding_bn_drop_cnn

'bash' ������گ�����⧻����  Ҳ����ǿ������ e ĳ����
�����������ļ����
```

In [2]:
```
#@title Configure Jupyter Notebook

import matplotlib
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

executed in 1.25s, finished 19:51:26 2023-09-28

Batch Normalization

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

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [1] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

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

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

In [12]:
```
# As usual, a bit of setup

import os
import time
import numpy as np
import matplotlib.pyplot as plt
from deeplearning.classifiers.fc_net import *
from deeplearning.data_utils import get_CIFAR10_data
from deeplearning.gradient_check import eval_numerical_gradient, eval_numerical_gra
from deeplearning.solver import Solver
import random
import torch
seed = 7
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

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

os.makedirs("submission_logs", exist_ok=True)

def abs_error(x, y):
    return np.max(np.abs(x - y))

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))))
```

executed in 243ms, finished 19:55:35 2023-09-28

In [5]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
    print ('%s: ' % k, v.shape)
```

executed in 4.27s, finished 19:51:44 2023-09-28

```
../../../cifar-10/cifar-10-batches-py\data_batch_1
../../../cifar-10/cifar-10-batches-py\data_batch_2
../../../cifar-10/cifar-10-batches-py\data_batch_3
../../../cifar-10/cifar-10-batches-py\data_batch_4
../../../cifar-10/cifar-10-batches-py\data_batch_5
../../../cifar-10/cifar-10-batches-py\test_batch
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,)
```

The forward propagation during training given input $\mathbf{X} \in \mathbb{R}^{n,d}$ is defined as:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{i,j}$$
$$\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(X_{i,j} - \mu_j)^2$$
$$Y_{i,j} = \mathrm{BN}(\mathbf{X}|\gamma,\beta)_{i,j} = \frac{X_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}\gamma_j + \beta_j$$

It would be helpful if you introduce another intermediate variable $\mathbf{Z} \in \mathbb{R}^{n,d}$:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{i,j}$$
$$\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(X_{i,j} - \mu_j)^2$$
$$Z_{i,j} = \frac{X_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$
$$Y_{i,j} = \mathrm{BN}(\mathbf{X}|\gamma,\beta)_{i,j} = Z_{i,j}\gamma_j + \beta_j$$

Question

Draw the computational graph of training-time batch normalization in your written

Batch normalization: Forward

In the file `deeplearning/layers.py` , implement the batch normalization forward pass in the function `batchnorm_forward` .

Don't forget to record batch statistics such as running mean/var during training.

During testing, μ, and σ^2 are running mean and variance that is previously recorded in the training process.

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

```
In  [4]: import numpy as np
         # 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
         N, D1, D2, D3 = 200, 50, 60, 3
         X = np.random.randn(N, D1)
         W1 = np.random.randn(D1, D2)
         W2 = np.random.randn(D2, D3)
         a = np.maximum(0, X.dot(W1)).dot(W2)

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

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

         # Now means should be close to beta and stds close to gamma
         gamma = np.asarray([1.0, 2.0, 3.0])
         beta = np.asarray([11.0, 12.0, 13.0])
         a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
         print ('After batch normalization (nontrivial gamma, beta)')
         print ('  means: ', a_norm.mean(axis=0))  # expected: (approx.) [11, 12, 13]
         print ('  stds: ', a_norm.std(axis=0))  # expected: (approx.) [1, 2, 3]

         executed in 151ms, finished 19:51:35 2023-09-28
```

```
Before batch normalization:
  means:  [-33.90168215  18.46269328  -7.49068667]
  stds:  [38.72112268 29.39668975 31.14272252]
After batch normalization (gamma=1, beta=0)
  mean:  [ 1.15463195e-16  2.00950367e-16 -1.11022302e-16]
  std:  [1.         0.99999999 0.99999999]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [1.         1.99999999 2.99999998]
```

In [6]:
```
# 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, D1, D2, D3 = 200, 50, 60, 3
np.random.seed(seed)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)
for t in range(50):
    X = np.random.randn(N, D1)
    a = np.maximum(0, X.dot(W1)).dot(W2)
    batchnorm_forward(a, gamma, beta, bn_param)
bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print ('After batch normalization (test-time):')
print ('  means: ', a_norm.mean(axis=0))
print ('  stds: ', a_norm.std(axis=0))
expected_a_norm = np.array(
    [[-7.37859885e-01,  2.10050591e+00, -3.24286480e-01],
    [ 2.02781031e+00,  1.92492178e-01,  1.54852388e+00],
    [ 5.44242949e-01,  1.07389911e+00,  8.06464618e-01],
    [-2.25599789e-02,  7.64501325e-01, -3.03045313e-01],
    [-9.74592587e-01,  6.01731799e-01, -6.57200019e-03]])
print ('Abs error of a_norm: ', abs_error(a_norm[:5, :], expected_a_norm))
```
executed in 410ms, finished 19:51:44 2023-09-28

```
After batch normalization (test-time):
  means:  [-0.00967681  0.01315673  0.00748036]
  stds:  [1.07009997 1.01651564 0.88995918]
Abs error of a_norm:  4.3424073226105975e-09
```

Batch Normalization: backward

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

To derive the backward pass you should refer to the computation graph for batch normalization and backprop through each of the intermediate nodes that you have drawn earlier. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Once you have finished, run the following to numerically check your backward pass.

In [7]:
```python
# Gradient check batchnorm backward pass

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 gamma: batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda beta: batchnorm_forward(x, gamma, beta, bn_param)[0]

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

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
print ('dx error: ', rel_error(dx_num, dx))
print ('dgamma error: ', rel_error(da_num, dgamma))
print ('dbeta error: ', rel_error(db_num, dbeta))
```

executed in 179ms, finished 19:51:48 2023-09-28

```
dx error:  2.0718950485957 24e-09
dgamma error:  2.3439059333191323e-12
dbeta error:  2.6356334174796573e-12
```

Batch Normalization: alternative backward (Optional)

There are two strategies to implement batch normalization of an operator consists of multiple parts:

1. Write out a computation graph composed of simple operations and backprop through all intermediate values. This is the general principal of automatic backpropagation in deep learning framework.
2. Work out the derivatives on paper. This usually applies to some operators to achieve better numerical stability or computational efficiency, such as `softmax + cross entropy` or `sigmoid + binary cross entropy`.

Surprisingly, it turns out that you can also derive a simple expression for the batch normalization backward pass if you work out derivatives on paper and simplify.

Question (Optional)

Derive the closed-form back-propagation of a batch normalization layer (during training). Include the answer in your written assignment.

Specifically, given $\mathrm{dy}_{i,j} = \frac{\partial \mathcal{L}}{\partial Y_{i,j}}$ for every i, j, Please derive $\frac{\partial \mathcal{L}}{\partial X_{i,j}}$ for every i, j as a function of $\mathrm{dy}, \mathbf{X}, \mu, \sigma^2, \epsilon, \gamma, \beta$.

After doing so (and additionally deriving $\frac{\partial \mathcal{L}}{\partial \gamma_j}$ and $\frac{\partial \mathcal{L}}{\partial \beta_j}$ for each j), implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

In [ ]:
```
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()
# repeat backwards passes multiple times for stability
for r in range(1000):
    dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
t2 = time.time()
for r in range(1000):
    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)))
```

Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `deeplearning/classifiers/fc_net.py` . Modify your implementation to add batch normalization.

Concretely, when the flag `use_batchnorm` is `True` 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 `deeplearning/layer_utils.py` . If you decide to do so, do it in the file `deeplearning/classifiers/fc_net.py` .

```
In [18]: N, D, H1, H2, C = 2, 15, 20, 30, 10

         model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                                   reg=0, weight_scale=5e-2, dtype=np.float64,
                                   use_batchnorm=True)
         model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H1).reshape(D, H1)
         model.params['b1'] = np.linspace(-0.1, 0.9, num=H1)
         model.params['W2'] = np.linspace(-0.3, 0.4, num=H1*H2).reshape(H1, H2)
         model.params['b2'] = np.linspace(-0.9, 0.1, num=H2)
         model.params['W3'] = np.linspace(-0.3, 0.4, num=H2*C).reshape(H2, C)
         model.params['b3'] = np.linspace(-0.9, 0.1, num=C)
         X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T

         expected_bn_forward_output = np.array([[0.28397701, 0.46532063, 0.64666426, 0.828007
                                                 1.37203875, 1.55338238,  1.734726, 1.916
                                                [-0.9, -0.78888889, -0.67777778, -0.56666667,
                                                 -0.23333333, -0.12222222, -0.01111111, (

         # Checks if initial forward pass is correct with batchnorm
         init_scores = model.loss(X)
         print('initial predictions error: %.2e' % rel_error(init_scores, expected_bn_forward

         X = np.random.randn(N, D)
         y = np.random.randint(C, size=(N,))

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


             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=
                 print ('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
             if reg == 0: print

executed in 3.68s, finished 20:03:13 2023-09-28
```

```
initial predictions error: 5.00e-08
Running check with reg =  0
Initial loss:  2.356793957612584
W1 relative error: 7.54e-05
W2 relative error: 2.47e-06
W3 relative error: 3.74e-10
b1 relative error: 2.22e-03
b2 relative error: 2.22e-03
b3 relative error: 1.70e-10
beta1 relative error: 1.17e-08
beta2 relative error: 1.13e-08
gamma1 relative error: 1.14e-08
gamma2 relative error: 2.80e-09
Running check with reg =  3.14
Initial loss:  6.978990270496991
W1 relative error: 1.79e-05
W2 relative error: 4.17e-05
W3 relative error: 4.31e-07
b1 relative error: 4.44e-03
b2 relative error: 2.66e-07
b3 relative error: 1.69e-10
beta1 relative error: 3.80e-09
beta2 relative error: 6.41e-08
gamma1 relative error: 3.96e-09
gamma2 relative error: 2.44e-08
```

Batchnorm for deep networks

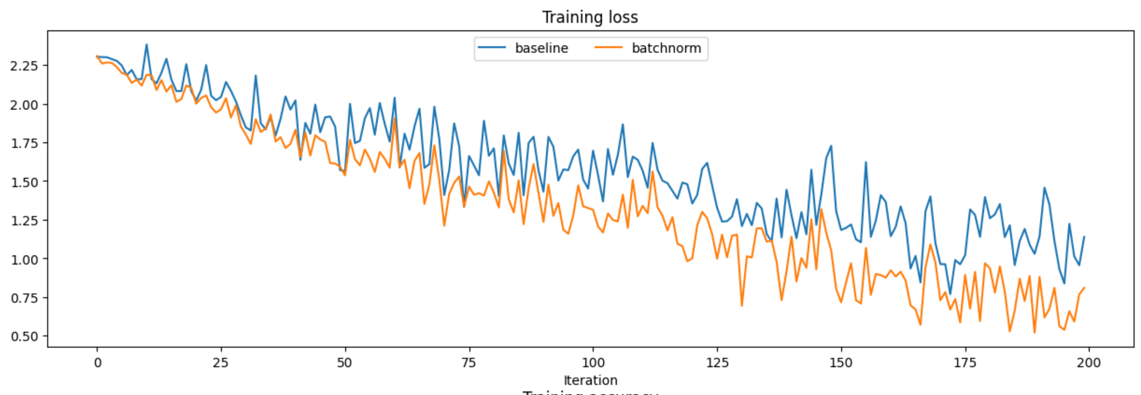
Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

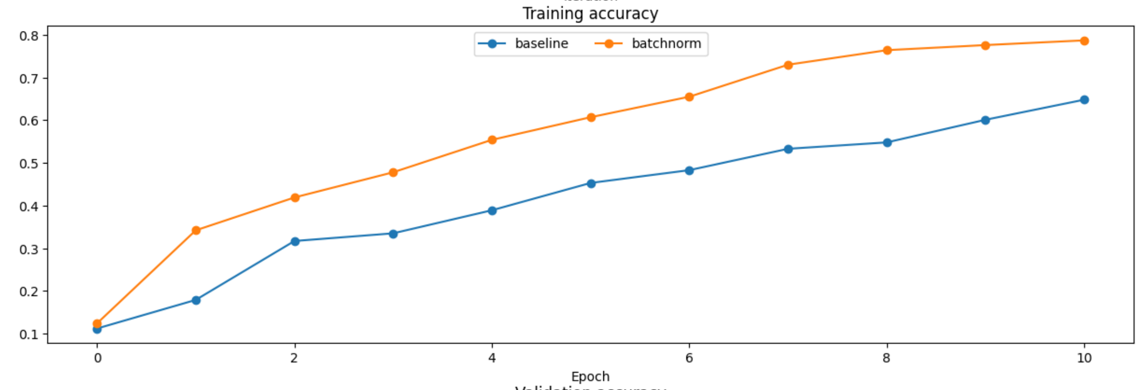
In [19]:
```
#debugging
from deeplearning.layers import *
from deeplearning.classifiers.fc_net import *

# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

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

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=T
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=Fal
np.random.seed(seed)
bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
bn_solver.train()

np.random.seed(seed)
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
solver.train()
```
executed in 56.7s, finished 20:04:14 2023-09-28

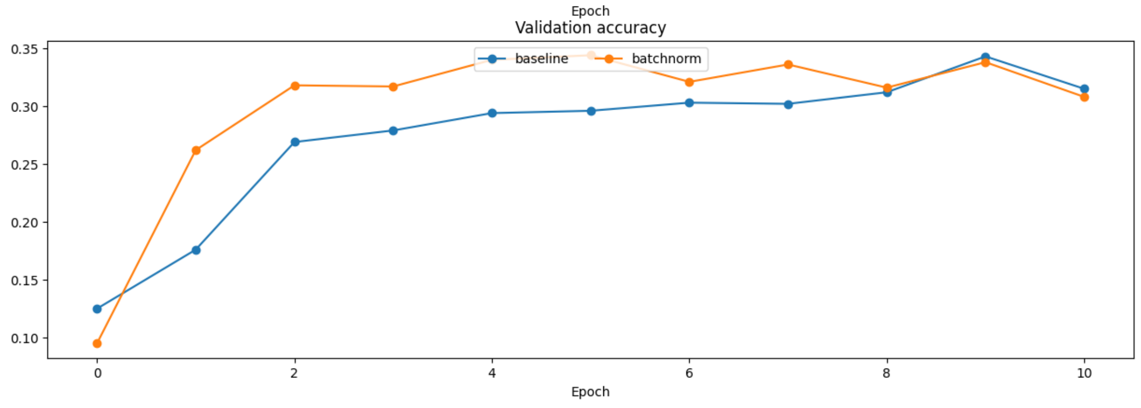
```
(Iteration 1 / 200) loss: 2.307697
(Epoch 0 / 10) train acc: 0.124000; val_acc: 0.095000
(Epoch 1 / 10) train acc: 0.342000; val_acc: 0.262000
(Epoch 2 / 10) train acc: 0.419000; val_acc: 0.318000
(Epoch 3 / 10) train acc: 0.478000; val_acc: 0.317000
(Epoch 4 / 10) train acc: 0.554000; val_acc: 0.340000
(Epoch 5 / 10) train acc: 0.607000; val_acc: 0.344000
(Epoch 6 / 10) train acc: 0.655000; val_acc: 0.321000
(Epoch 7 / 10) train acc: 0.730000; val_acc: 0.336000
(Epoch 8 / 10) train acc: 0.764000; val_acc: 0.316000
(Epoch 9 / 10) train acc: 0.776000; val_acc: 0.338000
(Epoch 10 / 10) train acc: 0.787000; val_acc: 0.308000
(Iteration 1 / 200) loss: 2.302319
(Epoch 0 / 10) train acc: 0.112000; val_acc: 0.125000
(Epoch 1 / 10) train acc: 0.179000; val_acc: 0.176000
(Epoch 2 / 10) train acc: 0.317000; val_acc: 0.269000
(Epoch 3 / 10) train acc: 0.335000; val_acc: 0.279000
(Epoch 4 / 10) train acc: 0.389000; val_acc: 0.294000
(Epoch 5 / 10) train acc: 0.453000; val_acc: 0.296000
(Epoch 6 / 10) train acc: 0.483000; val_acc: 0.303000
(Epoch 7 / 10) train acc: 0.533000; val_acc: 0.302000
(Epoch 8 / 10) train acc: 0.548000; val_acc: 0.312000
(Epoch 9 / 10) train acc: 0.601000; val_acc: 0.343000
(Epoch 10 / 10) train acc: 0.648000; val_acc: 0.315000
```

Run the following to visualize the results from two networks trained above and record the results of the experiment. You should find that using batch normalization helps the network to converge faster.

In [20]:
```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(solver.loss_history, '-', label='baseline')
plt.plot(bn_solver.loss_history, '-', label='batchnorm')

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

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

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

solver.record_histories_as_npz('submission_logs/compare_bn_deep_networks_no_bn.npz')
bn_solver.record_histories_as_npz('submission_logs/compare_bn_deep_networks_with_bn.
```
executed in 776ms, finished 20:05:48 2023-09-28







Batch normalization and initialization

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

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

In [21]:
```
# 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 = {}
solvers = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
    print ('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
    bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchno
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=

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

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

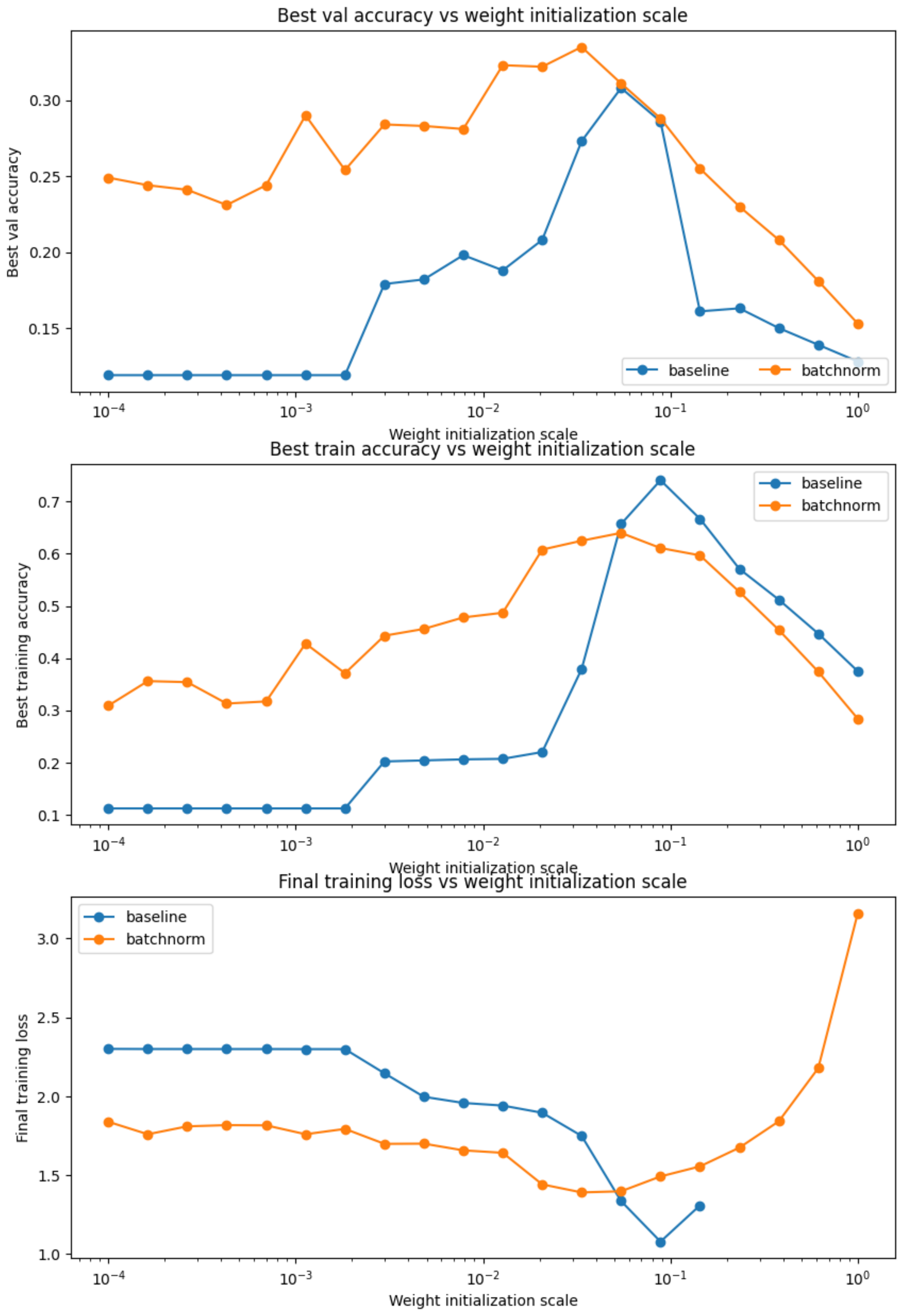
executed in 13m 7s, finished 20:19:52 2023-09-28

```
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
```

```
F:\new_gitee_code\berkeley_class\Deep_Learning\hw5\code\q_coding_bn_drop_cnn\deep
learning\layers.py:584: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N

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

In [22]:
```python
# 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 i, ws in enumerate(weight_scales):
    best_train_accs.append(max(solvers[ws].train_acc_history))
    bn_best_train_accs.append(max(bn_solvers[ws].train_acc_history))

    best_val_accs.append(max(solvers[ws].val_acc_history))
    bn_best_val_accs.append(max(bn_solvers[ws].val_acc_history))

    final_train_loss.append(np.mean(solvers[ws].loss_history[-100:]))
    bn_final_train_loss.append(np.mean(bn_solvers[ws].loss_history[-100:]))


    solvers[ws].record_histories_as_npz('submission_logs/bn_and_weight_scale_experim
    bn_solvers[ws].record_histories_as_npz('submission_logs/bn_and_weight_scale_expe

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

executed in 1.96s, finished 20:21:45 2023-09-28



Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features 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

Dropout forward pass

In the file `deeplearning/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. Input means should be approximately the same as the output means at both train/test time.

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

In [23]:
```
x = np.random.randn(500, 500) + 10

for p in [0.3, 0.6, 0.75]:
    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())
    # expected: (approx.) 10, 10, 10
    print ('Mean of train-time output: ', out.mean())
    # expected: (approx.) 10, 10, 10
    print ('Mean of test-time output: ', out_test.mean())
    # expected: (approx.) 10, 10, 10
    print ('Fraction of train-time output set to zero: ', (out == 0).mean())
    # expected: (approx.) 0.3, 0.6, 0.75
    print ('Fraction of test-time output set to zero: ', (out_test == 0).mean())
    # expected: (approx.) 0.0, 0.0, 0.0
```

executed in 161ms, finished 20:21:50 2023-09-28

```
Running tests with p =  0.3
Mean of input:  10.000349084992713
Mean of train-time output:  2.995196003346861
Mean of test-time output:  10.000349084992713
Fraction of train-time output set to zero:  0.70044
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  10.000349084992713
Mean of train-time output:  6.007020325435766
Mean of test-time output:  10.000349084992713
Fraction of train-time output set to zero:  0.39932
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  10.000349084992713
Mean of train-time output:  7.496448115088381
Mean of test-time output:  10.000349084992713
Fraction of train-time output set to zero:  0.250376
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

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

In [24]:
```
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, '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

print ('dx relative error: ', rel_error(dx, dx_num))
```
executed in 160ms, finished 20:22:40 2023-09-28

```
dx relative error:  1.8928972137129857e-11
```

Fully-connected nets with Dropout

In the file `deeplearning/classifiers/fc_net.py` , modify your implementation to use dropout. Specificially, if the constructor the the net receives a nonzero value for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

In [27]:
```
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 in [0, 0.25, 0.5]:
    print ('Running check with dropout = ', dropout)
    model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                              weight_scale=5e-2, dtype=np.float64,
                              dropout=dropout, seed=123)

    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=
        print ('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
    print
```

executed in 3.24s, finished 20:33:33 2023-09-28

```
Running check with dropout =  0
Initial loss:  2.3030442023102875
W1 relative error: 4.88e-07
W2 relative error: 7.72e-07
W3 relative error: 1.00e-07
b1 relative error: 6.59e-08
b2 relative error: 3.49e-08
b3 relative error: 1.32e-10
Running check with dropout =  0.25
Initial loss:  2.3023441248653347
W1 relative error: 1.95e-07
W2 relative error: 6.88e-09
W3 relative error: 1.05e-07
b1 relative error: 6.51e-09
b2 relative error: 6.38e-10
b3 relative error: 1.39e-10
Running check with dropout =  0.5
Initial loss:  2.305150366397604
W1 relative error: 1.08e-07
W2 relative error: 1.27e-07
W3 relative error: 3.63e-07
b1 relative error: 2.21e-09
b2 relative error: 2.85e-09
b3 relative error: 1.42e-10
```

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 dropout probability of 0.5. We will then visualize the training and validation accuracies of the two networks over time.

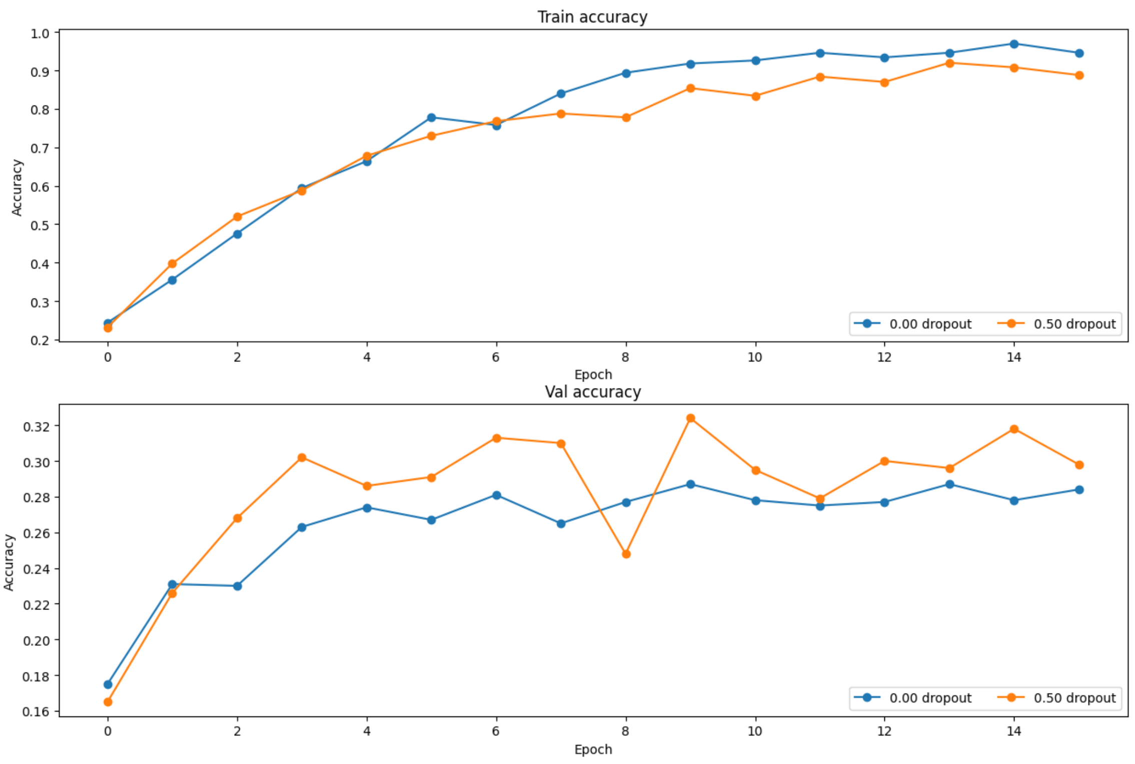
In [28]:
```
# Train two identical nets, one with dropout and one without

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 = [0, 0.5]
for dropout in dropout_choices:
    model = FullyConnectedNet([500], dropout=dropout)
    print (dropout)

    np.random.seed(seed)
    solver = Solver(model, small_data,
                    num_epochs=15, batch_size=100,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': 5e-4,
                    },
                    verbose=True, print_every=100)
    solver.train()
    solvers[dropout] = solver
```

executed in 15.1s, finished 20:36:10 2023-09-28

```
0
(Iteration 1 / 75) loss: 9.289189
(Epoch 0 / 15) train acc: 0.244000; val_acc: 0.175000

F:\new_gitee_code\berkeley_class\Deep_Learning\hw5\code\q_coding_bn_drop_cnn\deep
learning\layers.py:586: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
```

```
(Epoch 1 / 15) train acc: 0.356000; val_acc: 0.231000
(Epoch 2 / 15) train acc: 0.476000; val_acc: 0.230000
(Epoch 3 / 15) train acc: 0.594000; val_acc: 0.263000
(Epoch 4 / 15) train acc: 0.664000; val_acc: 0.274000
(Epoch 5 / 15) train acc: 0.778000; val_acc: 0.267000
(Epoch 6 / 15) train acc: 0.758000; val_acc: 0.281000
(Epoch 7 / 15) train acc: 0.840000; val_acc: 0.265000
(Epoch 8 / 15) train acc: 0.894000; val_acc: 0.277000
(Epoch 9 / 15) train acc: 0.918000; val_acc: 0.287000
(Epoch 10 / 15) train acc: 0.926000; val_acc: 0.278000
(Epoch 11 / 15) train acc: 0.946000; val_acc: 0.275000
(Epoch 12 / 15) train acc: 0.934000; val_acc: 0.277000
(Epoch 13 / 15) train acc: 0.946000; val_acc: 0.287000
(Epoch 14 / 15) train acc: 0.970000; val_acc: 0.278000
(Epoch 15 / 15) train acc: 0.946000; val_acc: 0.284000
0.5
(Iteration 1 / 75) loss: 6.038539
(Epoch 0 / 15) train acc: 0.232000; val_acc: 0.165000
(Epoch 1 / 15) train acc: 0.398000; val_acc: 0.226000
(Epoch 2 / 15) train acc: 0.520000; val_acc: 0.268000
(Epoch 3 / 15) train acc: 0.588000; val_acc: 0.302000
(Epoch 4 / 15) train acc: 0.678000; val_acc: 0.286000
(Epoch 5 / 15) train acc: 0.730000; val_acc: 0.291000
(Epoch 6 / 15) train acc: 0.768000; val_acc: 0.313000
(Epoch 7 / 15) train acc: 0.788000; val_acc: 0.310000
(Epoch 8 / 15) train acc: 0.778000; val_acc: 0.248000
(Epoch 9 / 15) train acc: 0.854000; val_acc: 0.324000
(Epoch 10 / 15) train acc: 0.834000; val_acc: 0.295000
(Epoch 11 / 15) train acc: 0.884000; val_acc: 0.279000
(Epoch 12 / 15) train acc: 0.870000; val_acc: 0.300000
(Epoch 13 / 15) train acc: 0.920000; val_acc: 0.296000
(Epoch 14 / 15) train acc: 0.908000; val_acc: 0.318000
(Epoch 15 / 15) train acc: 0.888000; val_acc: 0.298000
```

In [29]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
    solver = solvers[dropout]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])
    solver.record_histories_as_npz('submission_logs/dropout_regularization_experimen

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

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
    plt.plot(solvers[dropout].val_acc_history, '-o', label='%.2f dropout' % dropout)
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()
```
executed in 497ms, finished 20:36:31 2023-09-28



Question:

Explain what you see in this experiment. What does it suggest about dropout? Write your answer on the written assignment.

Setup Environment

If you are working on this assignment using Google Colab, please execute the codes below.

Alternatively, you can also do this assignment using a local anaconda environment (or a Python virtualenv). Please clone the GitHub repo by running `git clone https://github.com/Berkeley-CS182/cs182hw3.git` and refer to `README.md` for further details.

In [ ]:
```
#@title Mount your Google Drive

import os
from google.colab import drive
drive.mount('/content/gdrive')
```

In [ ]:
```
#@title Set up mount symlink

DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw3_sp23'
DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '')
if not os.path.exists(DRIVE_PYTHON_PATH):
  %mkdir $DRIVE_PATH

## the space in `My Drive` causes some issues,
## make a symlink to avoid this
SYM_PATH = '/content/cs182hw3'
if not os.path.exists(SYM_PATH):
  !ln -s $DRIVE_PATH $SYM_PATH
```

In [ ]:
```
#@title Install dependencies

!pip install numpy==1.21.6 imageio==2.9.0 matplotlib==3.2.2
```

In [ ]:
```
#@title Clone homework repo

%cd $SYM_PATH
if not os.path.exists("cs182hw3"):
  !git clone https://github.com/Berkeley-CS182/cs182hw3.git
%cd cs182hw3
```

In [ ]:
```
#@title Download datasets (Skip if you did it in the last part)

%cd deeplearning/datasets/
!bash ./get_datasets.sh
%cd ../..
```

In [1]:
```
#@title Configure Jupyter Notebook

import matplotlib
%matplotlib inline
%load_ext autoreload
%autoreload 2
```
executed in 1.22s, finished 20:43:12 2023-09-28

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.

In [1]:
```
# As usual, a bit of setup

import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import time
import numpy as np
import matplotlib.pyplot as plt
from deeplearning.classifiers.fc_net import *
from deeplearning.data_utils import get_CIFAR10_data
from deeplearning.gradient_check import eval_numerical_gradient, eval_numerical_gra
from deeplearning.solver import Solver
import random
import torch
seed = 7
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

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

os.makedirs("submission_logs", exist_ok=True)

def abs_error(x, y):
    return np.max(np.abs(x - y))

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))))
```
executed in 2.99s, finished 21:22:53 2023-09-29

In [2]:
```python
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
    print ('%s: ' % k, v.shape)
```

executed in 2.90s, finished 21:23:06 2023-09-29

```
../../../cifar-10/cifar-10-batches-py\data_batch_1
../../../cifar-10/cifar-10-batches-py\data_batch_2
../../../cifar-10/cifar-10-batches-py\data_batch_3
../../../cifar-10/cifar-10-batches-py\data_batch_4
../../../cifar-10/cifar-10-batches-py\data_batch_5
../../../cifar-10/cifar-10-batches-py\test_batch
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,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `deeplearning/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:

In [3]:
```
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 1e-8
print ('Testing conv_forward_naive')
print ('difference: ', rel_error(out, correct_out))
```

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

Convolution: naive backpropagation

In `deeplearning/layers.py` , implement the backpropagation for the convolution layer in the function `conv_backward_naive` .

The gradient check below will take 30s~1min depending on the efficiency of your code.

In [4]:
```
x = np.random.randn(10, 3, 5, 5)
w = np.random.randn(16, 3, 3, 3)
b = np.random.randn(16,)
conv_param = {'stride': 2, 'pad': 1}
dout = np.random.randn(10, 16, 3, 3)
out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: conv_forward_naive(xx, w, b, conv
dw_num = eval_numerical_gradient_array(lambda ww: conv_forward_naive(x, ww, b, conv
db_num = eval_numerical_gradient_array(lambda bb: conv_forward_naive(x, w, bb, conv

print ('dx relative error: ', rel_error(dx, dx_num))
print ('dw relative error: ', rel_error(dw, dw_num))
print ('db relative error: ', rel_error(db, db_num))
```

executed in 55.6s, finished 21:35:48 2023-09-28

```
dx relative error:  5.742406157093703e-07
dw relative error:  9.391851964541804e-09
db relative error:  9.784882150201716e-11
```

Max pooling: Naive forward

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

Check your implementation by running the following:

In [7]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_height': 2, 'pool_width': 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 around 1e-8.
print ('Testing max_pool_forward_naive function:')
print ('difference: ', rel_error(out, correct_out))
```
executed in 313ms, finished 01:42:46 2023-09-29

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

Max pooling: Naive backward

In `deeplearning/layers.py` , implement the backpropagation for the max pooling layer in the function `max_pool_backward_naive` .

In [8]:
```
x = np.random.randn(10, 3, 8, 7)
pool_param = {'pool_height': 2, 'pool_width': 3, 'stride': 2}
dout = np.random.randn(10, 3, 4, 3)
out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: max_pool_forward_naive(xx, pool_p

print ('dx relative error: ', rel_error(dx, dx_num))
```
executed in 11.4s, finished 01:43:05 2023-09-29

```
dx relative error:  3.2760907074267422e-12
```

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `deeplearning/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

The gradient check below will take 45s~1min30s depending on the efficiency of your code.

In [9]:
```
from deeplearning.layer_utils import conv_relu_pool_forward, conv_relu_pool_backwar

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, co
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, co
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, co

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))
```

executed in 1m 24.4s, finished 11:48:54 2023-09-29

```
Testing conv_relu_pool
dx error:  5.859573170575022e-08
dw error:  1.4096284487344332e-09
db error:  2.1855649049573366e-11
```

Three-layer ConvNet

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

Open the file `deeplearning/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Run the following cells to help you debug:

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 this should go up.

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.

In [2]:
```
from deeplearning.classifiers.cnn import ThreeLayerConvNet

np.random.seed(seed)
model = ThreeLayerConvNet(num_filters=3, filter_size=1)

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)
# expected: (approx.) 2.302585092994046

model.reg = 0.5
loss, grads = model.loss(X, y)
print ('Initial loss (with regularization): ', loss)
# expected: (approx.) 2.322037342994046
```
execution queued 21:23:17 2023-09-29

```
Initial loss (no regularization):  2.302585138003613
Initial loss (with regularization):  2.3218146990940824
```

The following gradient check will take 1min30s to 3min to run. The max relative error of every parameter tensor should be less than `1e-2` .

In [3]:
```
num_inputs = 5
input_dim = (3, 12, 12)
reg = 0.0
num_classes = 10
np.random.seed(seed)
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,
                          weight_scale=0.01, reg=0.001, dtype=np.float64)
loss, grads = model.loss(X, y)
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=Fa
    e = rel_error(param_grad_num, grads[param_name])
    print ('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grad
```

```
W1 max relative error: 2.894889e-05
W2 max relative error: 1.827179e-03
W3 max relative error: 1.941106e-03
b1 max relative error: 9.828774e-07
b2 max relative error: 4.721508e-08
b3 max relative error: 1.270831e-09
```

Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. 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 the statistics of each feature channel to be relatively consistent both between different imagesand different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

Spatial batch normalization: forward

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

In [3]:
```python
# 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. Shape should be unchanged.
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. Shape should be unchnaged.
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:  [ 9.53942646 10.42457703 10.40605234]
  Stds:  [3.77505361 4.29892071 3.81059736]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [-1.22124533e-16 -2.35922393e-16 -4.60742555e-16]
  Stds:  [0.99999965 0.99999973 0.99999966]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999895 3.99999892 4.99999828]
```

In [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.

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:  [-0.02549464  0.0306877  -0.01219379  0.03943168]
  stds:  [0.96438738 0.98319165 1.01579818 1.02752983]
```

Spatial batch normalization: backward

In the file `deeplearning/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:

In [3]:
```
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)

_, 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))
```

```
dx error:  9.397350278853868e-08
dgamma error:  1.6257023108091236e-11
dbeta error:  5.6908669881601136e-11
```

In [ ]:
```
```

Setup Environment

If you are working on this assignment using Google Colab, please execute the codes below.

In [ ]:
```
#@title Mount your Google Drive

import os
from google.colab import drive
drive.mount('/content/gdrive')
```

In [ ]:
```
#@title Set up mount symlink

DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw3_sp23'
DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '')
if not os.path.exists(DRIVE_PYTHON_PATH):
  %mkdir $DRIVE_PATH

## the space in `My Drive` causes some issues,
## make a symlink to avoid this
SYM_PATH = '/content/cs182hw3'
if not os.path.exists(SYM_PATH):
  !ln -s $DRIVE_PATH $SYM_PATH
```

In [ ]:
```
#@title Install dependencies

!pip install numpy==1.21.6 imageio==2.9.0 matplotlib==3.2.2
```

In [ ]:
```
#@title Clone homework repo

%cd $SYM_PATH
if not os.path.exists("cs182hw3"):
  !git clone https://github.com/Berkeley-CS182/cs182hw3.git
%cd cs182hw3
```

In [19]:
```
#@title Configure Jupyter Notebook

import matplotlib
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

executed in 224ms, finished 12:53:43 2023-09-30

Train Convolutional Neural Networks using PyTorch

In this notebook we will put everything together you've learned: affine layers, relu layers, conv layers, max-pooling, (spatial) batch norm, and dropout, and train CNNs on CIFAR-100.

However, our implementation of these modules in NumPy are quite inefficient---especially convolutional layers. Therefore, we use PyTorch with GPU in this coding assignment.

Make sure you have access to GPUs when running this notebook. On Google Colab, you can switch to a GPU runtime by clicking "Runtime" - "Change Runtime Type" - "GPU" in the menu on the top of the webpage.

In [14]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import json
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils as utils
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

os.makedirs("submission_logs", exist_ok=True)
```
executed in 7ms, finished 12:51:48 2023-09-30

In [2]:
```
torch.cuda.is_available()
# make sure GPU is enabled
```
executed in 56ms, finished 12:47:22 2023-09-30

Out[2]:
```
True
```

In [3]:
```
seed = 227
```
executed in 7ms, finished 12:47:23 2023-09-30

Load and Visualize Data

In this cell, we load and visualize the CIFAR100 dataset. Note that we apply data augmentation (random horizontal flip) to the training dataset:

```
transforms.RandomHorizontalFlip()
```

Data augmentation is a popular technique in machine learning and computer vision that involves generating additional training data to improve the performance of a model. One common form of data augmentation for image data is random horizontal flipping, which involves flipping an image horizontally with a 50% chance during training. This technique is often used to increase the variability of the training data and to help the model generalize better to new, unseen images. By randomly flipping images, the model is exposed to a wider range of orientations and can better learn to recognize features that are invariant to horizontal flipping.

In [4]:
```
valid_test_transform = transforms.Compose(
    [
        transforms.ToTensor(),   # convert image to PyTorch Tensor
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        # normalize to [-1.0, 1.0] (originally [0.0, 1.0])
    ]
)

train_transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        transforms.RandomHorizontalFlip()   # data augmentation
    ]
)

# Download training data from open datasets.
training_data = datasets.CIFAR100(
    root="../../../cifar-100",
    train=True,
    download=True,
    transform=train_transform,
)

# Download test data from open datasets.
valid_test_data = datasets.CIFAR100(
    root="../../../cifar-100",
    train=False,
    download=True,
    transform=valid_test_transform,
)

# split original test data to valid data and test data
valid_data = list(valid_test_data)[::2]
test_data = list(valid_test_data)[1::2]

classes = [
    "apple",
    "aquarium_fish",
    "baby",
    "bear",
    "beaver",
    "bed",
    "bee",
    "beetle",
    "bicycle",
    "bottle",
    "bowl",
    "boy",
    "bridge",
    "bus",
    "butterfly",
    "camel",
    "can",
    "castle",
    "caterpillar",
    "cattle",
    "chair",
    "chimpanzee",
    "clock",
    "cloud",
    "cockroach",
    "couch",
    "cra",
    "crocodile",
    "cup",
    "dinosaur",
    "dolphin",
    "elephant",
    "flatfish",
    "forest",
    "fox",
    "girl",
    "hamster",
    "house",
    "kangaroo",
    "keyboard",
    "lamp",
    "lawn_mower",
    "leopard",
    "lion",
    "lizard",
    "lobster",
    "man",
    "maple_tree",
    "motorcycle",
    "mountain",
    "mouse",
    "mushroom",
    "oak_tree",
    "orange",
    "orchid",
    "otter",
    "palm_tree",
    "pear",
    "pickup_truck",
    "pine_tree",
    "plain",
    "plate",
    "poppy",
    "porcupine",
    "possum",
    "rabbit",
    "raccoon",
    "ray",
    "road",
    "rocket",
    "rose",
    "sea",
    "seal",
    "shark",
    "shrew",
    "skunk",
    "skyscraper",
    "snail",
    "snake",
    "spider",
    "squirrel",
    "streetcar",
    "sunflower",
    "sweet_pepper",
    "table",
    "tank",
    "telephone",
    "television",
    "tiger",
    "tractor",
    "train",
    "trout",
    "tulip",
    "turtle",
    "wardrobe",
    "whale",
    "willow_tree",
    "wolf",
    "woman",
    "worm",
]
```

executed in 10.9s, finished 12:47:36 2023-09-30

```
Files already downloaded and verified
Files already downloaded and verified
```

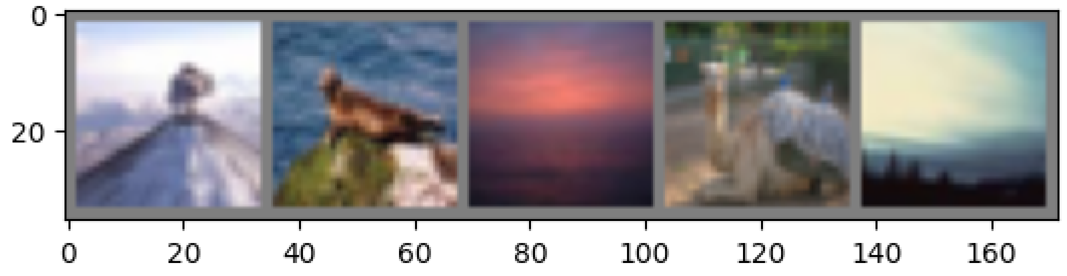
In [5]:
```
# Create data loaders.
valid_dataloader = utils.data.DataLoader(valid_data, batch_size=5)

for X, y in valid_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

executed in 17ms, finished 12:47:36 2023-09-30

```
Shape of X [N, C, H, W]: torch.Size([5, 3, 32, 32])
Shape of y: torch.Size([5]) torch.int64
```

Here is a visualization of 5 images in the validation dataset:

```
In [7]: # functions to show an image


        def imshow(img):
            img = img / 2 + 0.5     # unnormalize
            npimg = img.numpy()
            plt.imshow(np.transpose(npimg, (1, 2, 0)))
            plt.show()


        # get some random training images
        dataiter = iter(valid_dataloader)
        images, labels = next(dataiter)

        # show images
        imshow(torchvision.utils.make_grid(images))
        # print labels
        print('   '.join(f'{classes[labels[j]]:5s}' for j in range(5)))
```



```
mountain   seal    sea     camel   cloud
```

Define the Neural Network Architecture

Complete the code in `dl_pytorch/model.py` to finish the implementation of a convolutional neural network with batch normalization and dropout.

```
In [6]: from dl_pytorch.model import NeuralNetwork

        model = NeuralNetwork()
        print(model)

        assert len(model.state_dict()) == 10
        assert model.conv1.weight.shape == torch.Size([16, 3, 3, 3])
        assert model.conv1.bias.shape == torch.Size([16])
        assert model.conv2.weight.shape == torch.Size([32, 16, 3, 3])
        assert model.conv2.bias.shape == torch.Size([32])
        assert model.conv3.weight.shape == torch.Size([64, 32, 3, 3])
        assert model.conv3.bias.shape == torch.Size([64])
        assert model.fc1.weight.shape == torch.Size([256, 1024])
        assert model.fc1.bias.shape == torch.Size([256])
        assert model.fc2.weight.shape == torch.Size([100, 256])
        assert model.fc2.bias.shape == torch.Size([100])
        assert model(torch.randn(9, 3, 32, 32)).shape == torch.Size([9, 100])

        model = NeuralNetwork(do_batchnorm=True, p_dropout=0.1)
        assert len(model.state_dict()) == 25
        assert model.bn1.weight.shape == model.bn1.bias.shape == torch.Size([16])
        assert model.bn2.weight.shape == model.bn2.bias.shape == torch.Size([32])
        assert model.bn3.weight.shape == model.bn3.bias.shape == torch.Size([64])
        assert model(torch.randn(11, 3, 32, 32)).shape == torch.Size([11, 100])
```

executed in 145ms, finished 12:47:39 2023-09-30

```
NeuralNetwork(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (relu3): ReLU()
  (fc1): Linear(in_features=1024, out_features=256, bias=True)
  (relu4): ReLU()
  (fc2): Linear(in_features=256, out_features=100, bias=True)
)
```

Train the Neural Network

Complete the code cells below to train your neural network.

In [7]:
```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.cuda(), y.cuda()

        pred = model(X)
        loss = loss_fn(pred, y)

        ########################################################################
        # TODO: complete the following code for backpropagation and gradient
        #  update of a single step.
        # Hint: 3 lines
        ########################################################################
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ########################################################################

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```
executed in 14ms, finished 12:47:43 2023-09-30

In [8]:
```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.cuda(), y.cuda()
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Evaluation Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_l
    return 100*correct
```
executed in 16ms, finished 12:47:47 2023-09-30

In [9]:
```
def get_optimizer(params, optim_type, lr, momentum, lr_decay, l2_reg):
    if optim_type == "sgd":
        optimizer = optim.SGD(params, lr=lr, momentum=0.0, weight_decay=l2_reg)
    elif optim_type == "sgd_momentum":
        optimizer = optim.SGD(params, lr=lr, momentum=momentum,
                              weight_decay=l2_reg)
    elif optim_type == "adam":
        optimizer = optim.AdamW(params, lr=lr, betas=(momentum, 0.999),
                                weight_decay=l2_reg)
    else:
        raise ValueError(optim_type)
    scheduler = optim.lr_scheduler.ExponentialLR(optimizer, lr_decay)
    return optimizer, scheduler
```
executed in 15ms, finished 12:47:49 2023-09-30

Train the neural network. It should achieve at least 35% accuracy on the test set.

In [10]:
```python
def run_training(hp, nn_cls, save_prefix):
    print("Hyperparameters:", hp)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)

    model = nn_cls(do_batchnorm=hp.do_batchnorm, p_dropout=hp.p_dropout).cuda()

    # Create data loaders.
    train_dataloader = utils.data.DataLoader(
        training_data, batch_size=hp.batch_size)
    valid_dataloader = utils.data.DataLoader(
        valid_data, batch_size=hp.batch_size)

    loss_fn = nn.CrossEntropyLoss()
    optimizer, scheduler = get_optimizer(
        model.parameters(), hp.optim_type, hp.lr, hp.momentum, hp.lr_decay,
        hp.l2_reg)

    for t in range(hp.epochs):
        print(f"Epoch {t+1}\n-------------------------------")
        train(train_dataloader, model, loss_fn, optimizer)
        test(valid_dataloader, model, loss_fn)
        scheduler.step()


    print(f"Saving the model to submission_logs/{save_prefix}.pt")
    torch.save(model.state_dict(), f"submission_logs/{save_prefix}.pt")
    return model

def eval_on_test(hp, model, save_prefix):
    train_dataloader = utils.data.DataLoader(
        training_data, batch_size=hp.batch_size)
    test_dataloader = utils.data.DataLoader(
        test_data, batch_size=hp.batch_size)
    loss_fn = nn.CrossEntropyLoss()
    print("Evaluating on the test set")
    test_acc = test(test_dataloader, model, loss_fn)
    n_params = sum(p.numel() for p in model.parameters())
    print("Parameter count: {}".format(n_params))
    n_steps = len(train_dataloader) * hp.epochs
    print("Training steps: {}".format(n_steps))
    with open(f"submission_logs/{save_prefix}.json", "w", encoding="utf-8") as f:
        json.dump({
            "test_acc": test_acc,
            "hparams": hp.__dict__,
            "n_params": n_params,
            "n_steps": n_steps
        }, f)
```
executed in 22ms, finished 12:47:52 2023-09-30

In [13]:
```python
from dl_pytorch.hparams import HP as hp_base

model = run_training(hp_base, NeuralNetwork, "model")
eval_on_test(hp_base, model, "model")
```

```
loss: 4.500916  [16000/50000]
loss: 3.487213  [19200/50000]
loss: 3.989523  [22400/50000]
loss: 3.612078  [25600/50000]
loss: 3.594875  [28800/50000]
loss: 3.637436  [32000/50000]
loss: 3.702832  [35200/50000]
loss: 3.553779  [38400/50000]
loss: 3.550426  [41600/50000]
loss: 2.856360  [44800/50000]
loss: 3.210541  [48000/50000]
Evaluation Error:
 Accuracy: 19.2%, Avg loss: 3.346984

Epoch 2
-------------------------------
loss: 3.489315  [    0/50000]
loss: 3.505857  [ 3200/50000]
loss: 3.643275  [ 6400/50000]
```

Train the neural network with batch norm and dropout. It should achieve at least 38% accuracy on the test set.

In [12]:
```python
from dl_pytorch.hparams_bn_drop import HP as hp_bn_drop

model = run_training(hp_bn_drop, NeuralNetwork, "model_bn_drop")
eval_on_test(hp_bn_drop, model, "model_bn_drop")
```

```
loss: 2.514632  [22400/50000]
loss: 2.283799  [25600/50000]
loss: 2.255618  [28800/50000]
loss: 2.404037  [32000/50000]
loss: 2.473596  [35200/50000]
loss: 2.299351  [38400/50000]
loss: 2.547761  [41600/50000]
loss: 2.227181  [44800/50000]
loss: 2.302718  [48000/50000]
Evaluation Error:
 Accuracy: 38.0%, Avg loss: 2.361276

Saving the model to submission_logs/model_bn_drop.pt
Evaluating on the test set
Evaluation Error:
 Accuracy: 39.2%, Avg loss: 2.333904

Parameter count: 311908
Training steps: 7815
```

Design your own neural network

It's time to showcase your deep learning skills! In this assignment, you will be designing your own neural network using PyTorch. Your task is to **implement your neural network design in the files** `dl_pytorch/my_model.py` **and** `dl_pytorch/hparams_my_model.py` . The goal is to achieve a test accuracy of **44%** or higher.

To ensure reproducibility and to maintain the focus of the assignment, please adhere to the following rules:

1. Do not modify the code in the Jupyter Notebook cell or other cells that this cell depends on. It means that you cannot change data processing, the training loop, and the random seed. The emphasis of this assignment is on the model architecture and hyperparameter tuning.
2. The number of model parameters must not exceed $1,000,000$.
3. The total number of training steps should be no more than $20,000$.
4. The maximum number of training epochs is 10 .
5. Please refrain from using any pre-trained models or other downloaded assets.

Your test accuracy will be displayed on the Gradescope leaderboard. Please note that your rank on the leaderboard does not affect your grade. In order to receive full credit for this part of the assignment, you only need to abide by the rules outlined above and achieve a minimum test accuracy of 44%. Your grade will be scaled linearly, with a score of 0 for a test accuracy of 38% and full credit for a test accuracy of 44% or higher.

In [21]:
```
from dl_pytorch.my_model import MyNeuralNetwork
from dl_pytorch.hparams_my_model import HP as hp_my_model
```

executed in 208ms, finished 12:53:51 2023-09-30

In [33]:
```python
# without dropout

from dl_pytorch.my_model import MyNeuralNetwork
from dl_pytorch.hparams_my_model import HP as hp_my_model
model = run_training(hp_my_model, MyNeuralNetwork, "model_my_model")
```

executed in 5m 27s, finished 13:30:34 2023-09-30

```
loss: 1.873196  [ 3200/50000]
loss: 2.353914  [ 6400/50000]
loss: 1.903334  [ 9600/50000]
loss: 1.370685  [12800/50000]
loss: 1.439958  [16000/50000]
loss: 1.834635  [19200/50000]
loss: 1.993013  [22400/50000]
loss: 1.833073  [25600/50000]
loss: 1.444873  [28800/50000]
loss: 1.694107  [32000/50000]
loss: 2.021711  [35200/50000]
loss: 1.321993  [38400/50000]
loss: 1.760743  [41600/50000]
loss: 0.957302  [44800/50000]
loss: 1.521369  [48000/50000]
Evaluation Error:
 Accuracy: 42.4%, Avg loss: 2.309750

Epoch 6
```

In [34]:
```python
# with dropout

from dl_pytorch.my_model import MyNeuralNetwork
from dl_pytorch.hparams_my_model import HP as hp_my_model
model = run_training(hp_my_model, MyNeuralNetwork, "model_my_model")
```

executed in 5m 45s, finished 13:37:03 2023-09-30

```
loss: 1.769822  [ 3200/50000]
loss: 1.435440  [ 6400/50000]
loss: 1.694676  [ 9600/50000]
loss: 1.271162  [12800/50000]
loss: 1.523520  [16000/50000]
loss: 1.458426  [19200/50000]
loss: 1.866460  [22400/50000]
loss: 1.311532  [25600/50000]
loss: 1.317868  [28800/50000]
loss: 1.152032  [32000/50000]
loss: 1.548461  [35200/50000]
loss: 1.166089  [38400/50000]
loss: 1.074004  [41600/50000]
loss: 0.697782  [44800/50000]
loss: 1.191691  [48000/50000]
Evaluation Error:
 Accuracy: 45.1%, Avg loss: 2.191950

Saving the model to submission_logs/model_my_model.pt
```

In [35]:
```
# When you are ready to eval on test set, run this cell
# WARNING: In real-world applications, it is a bad practice to evaluate
#          frequently on the test set because the model will then perform poorly
#          on new, unseen data even if it achieves a high test accuracy.
eval_on_test(hp_my_model, model, "model_my_model")
```

executed in 1.24s, finished 13:37:26 2023-09-30

```
Evaluating on the test set
Evaluation Error:
 Accuracy: 46.0%, Avg loss: 2.164268

Parameter count: 11025924
Training steps: 12504
```

Question:

Briefly describe your neural network design and the procedure of hyperparameter tuning. Please include the answer of this question in your written assignment.

Collect your submissions

The following command will collect your solutions generated by both notebooks.

On Colab, after running the following cell, you can download your submissions from the `Files` tab, which can be opened by clicking the file icon on the left hand side of the screen.

In [ ]:
```
!rm -f cs182hw3_submission.zip
!zip -r cs182hw3_submission.zip . -x "*.git*" "*deeplearning/datasets*" "data*" "*.
```