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]:
```
# 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:  2.185564904957366e-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]:
```
# 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 [ ]: