

FullyConnectedNets

April 25, 2021

```
[1]: # this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = 'computer vision/assignments/assignment2'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/computer
vision/assignments/assignment2/CV7062610/datasets
/content
```

1 Fully-Connected Neural Nets

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

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a forward and a backward function. The forward function will

receive inputs, weights, and other parameters and will return both an output and a cache object storing data needed for the backward pass, like this:

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

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

    return out, cache
```

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

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

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

    return dx, dw
```

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

In addition to implementing fully-connected networks of arbitrary depth,(in the second part of the assingment) we will explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

```
[2]: # As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from CV7062610.classifiers.fc_net import *
from CV7062610.data_utils import get_CIFAR10_data
from CV7062610.gradient_check import eval_numerical_gradient, \
    eval_numerical_gradient_array
from CV7062610.solver import Solver

%matplotlib inline
```

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

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

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

[3]: # Load the (preprocessed) CIFAR10 data.

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

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

2 Affine layer (fully-connected): forward

Open the file CV7062610/layers.py and implement the affine_forward function.

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

[4]: # Test the affine_forward function

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

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

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

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
```

```
[ 3.25553199,  3.5141327,  3.77273342]])
```

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

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

3 Affine layer: backward

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

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

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

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

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

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

4 ReLU activation: forward

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

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

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

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

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

Testing relu_forward function:
difference: 4.999999798022158e-08

5 ReLU activation: backward

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

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

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

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

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

Testing relu_backward function:
dx error: 3.2756349136310288e-12

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

5.2 Answer:

the sigmoid function. when the values of the function is too high or too low the derivative becomes very small, and could lead to getting zero gradient during the backpropagation process.

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file CV7062610/layer_utils.py.

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

```
[8]: from CV7062610.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

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

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

Testing affine_relu_forward and affine_relu_backward:

```
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax

You implemented this loss functions in the last assignment, so we'll give it to you for free here. You should still make sure you understand how it work by looking at the implementations in CV7062610/layers.py.

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

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

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

# Test softmax_loss function. Loss should be close to 2.3 and dx error should
    be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

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

8 Two-layer network

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

Open the file CV7062610 /classifiers/fc_net.py and complete the implementation of the TwoLayerNet class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

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

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

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

```

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
    [[11.53165108, 12.2917344, 13.05181771, 13.81190102, 14.57198434, 15.
    →33206765, 16.09215096],
    [12.05769098, 12.74614105, 13.43459113, 14.1230412, 14.81149128, 15.
    →49994135, 16.18839143],
    [12.58373087, 13.20054771, 13.81736455, 14.43418138, 15.05099822, 15.
    →66781506, 16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

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

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

    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
        print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))

```

```

Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09

```



```

b2 relative error: 4.33e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10

```

9 Solver

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

Open the file `CV7062610/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least 45% accuracy on the validation set. If you are doing it correctly this shouldn't take more than 5 epochs of training and more than 5 minutes using vanilla `sgd`.

```

[11]: model = TwoLayerNet()
      solver = None
      best_val = -1
      #####
      # TODO: Use a Solver instance to train a TwoLayerNet that achieves at least #
      # 50% accuracy on the validation set.                                     #
      #####
      # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

      solver = Solver(model, data,
                      update_rule='sgd',
                      optim_config={
                          'learning_rate': 1e-3,
                      },
                      lr_decay=0.95,
                      num_epochs=5, batch_size=100,
                      print_every=100)
      solver.train()

      pass

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

```

```

(Iteration 1 / 2450) loss: 2.305171
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.085000
(Iteration 101 / 2450) loss: 1.868486
(Iteration 201 / 2450) loss: 1.932779

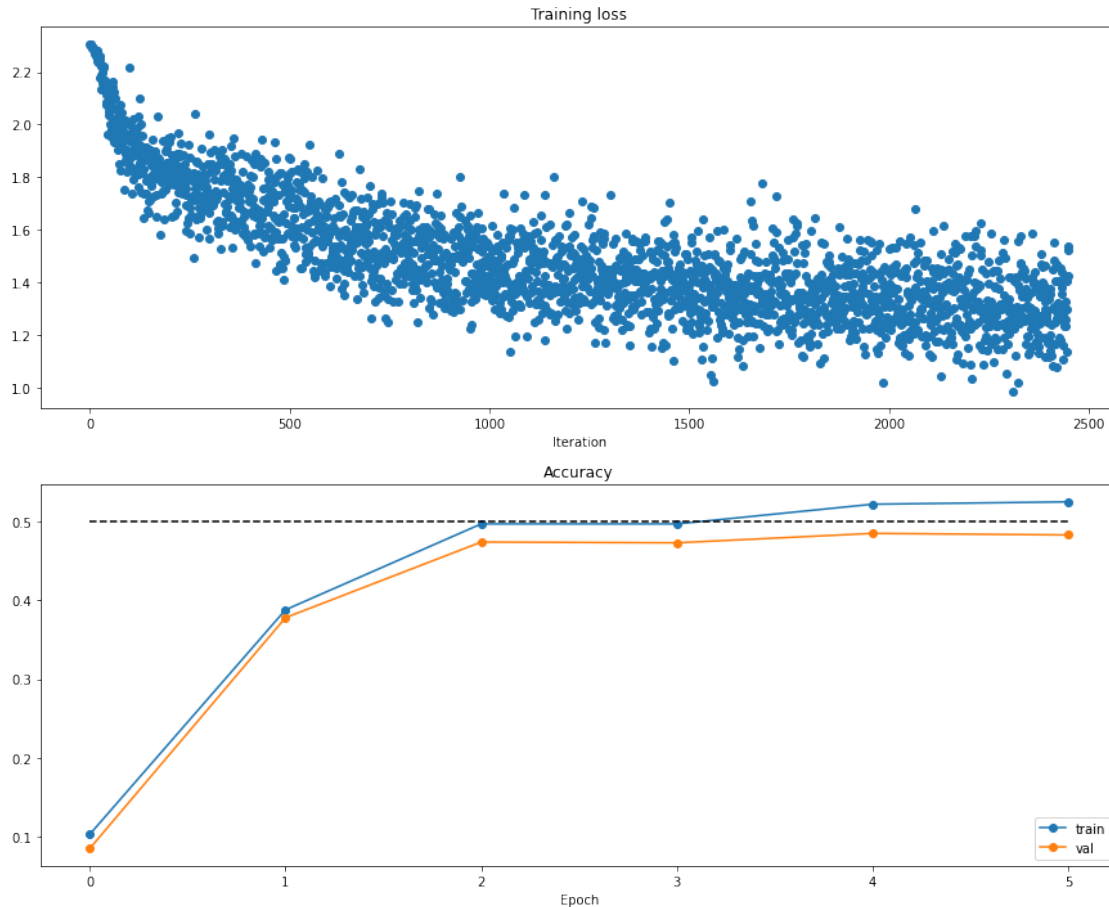
```

```
(Iteration 301 / 2450) loss: 1.962949
(Iteration 401 / 2450) loss: 1.538737
(Epoch 1 / 5) train acc: 0.388000; val_acc: 0.378000
(Iteration 501 / 2450) loss: 1.590466
(Iteration 601 / 2450) loss: 1.701701
(Iteration 701 / 2450) loss: 1.770164
(Iteration 801 / 2450) loss: 1.699713
(Iteration 901 / 2450) loss: 1.480728
(Epoch 2 / 5) train acc: 0.497000; val_acc: 0.474000
(Iteration 1001 / 2450) loss: 1.424826
(Iteration 1101 / 2450) loss: 1.323088
(Iteration 1201 / 2450) loss: 1.666793
(Iteration 1301 / 2450) loss: 1.445108
(Iteration 1401 / 2450) loss: 1.211955
(Epoch 3 / 5) train acc: 0.497000; val_acc: 0.473000
(Iteration 1501 / 2450) loss: 1.373108
(Iteration 1601 / 2450) loss: 1.281926
(Iteration 1701 / 2450) loss: 1.225324
(Iteration 1801 / 2450) loss: 1.357318
(Iteration 1901 / 2450) loss: 1.350007
(Epoch 4 / 5) train acc: 0.522000; val_acc: 0.485000
(Iteration 2001 / 2450) loss: 1.311724
(Iteration 2101 / 2450) loss: 1.381113
(Iteration 2201 / 2450) loss: 1.228021
(Iteration 2301 / 2450) loss: 1.290683
(Iteration 2401 / 2450) loss: 1.336943
(Epoch 5 / 5) train acc: 0.525000; val_acc: 0.483000
```

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

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

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



10 Multilayer network

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

Read through the `FullyConnectedNet` class in the file `CV7062610/classifiers/fc_net.py`.

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

10.1 Initial loss and gradient check

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

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

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

```

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

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

    # Most of the errors should be on the order of e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of e-5
    # for the check when reg = 0.0
    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,
        ↪h=1e-5)
        print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))

```

```

Running check with reg = 0
Initial loss: 0.0
Running check with reg = 3.14
Initial loss: 0.0

```

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

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

```

num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

##### TODO: find the best lr and wight scale #####
weight_scale = 1e-2    # Experiment with this!
learning_rate = 1e-5   # Experiment with this!
#####
model = FullyConnectedNet([100, 100],
                          weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={

```

```

        'learning_rate': learning_rate,
    }
)
solver.train()

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

```

(Iteration 1 / 40) loss: 0.000000

```

└──────────────────────────────────────────────────────────────────────────────────┘
└──────────────────────────────────────────────────────────────────────────────────┘

AxisError                                Traceback (most recent call└
last)

<ipython-input-15-80cfe57e2cf6> in <module>()
    23         }
    24     )
--> 25 solver.train()
    26
    27 plt.plot(solver.loss_history, 'o')

/content/drive/My Drive/computer vision/assignments/assignment2/
└CV7062610/solver.py in train(self)
    284         if first_it or last_it or epoch_end:
    285             train_acc = self.check_accuracy(
--> 286                 self.X_train, self.y_train, num_samples=self.
└num_train_samples
    287             )
    288             val_acc = self.check_accuracy(

/content/drive/My Drive/computer vision/assignments/assignment2/
└CV7062610/solver.py in check_accuracy(self, X, y, num_samples, batch_size)
    246         end = (i + 1) * batch_size
    247         scores = self.model.loss(X[start:end])
--> 248         y_pred.append(np.argmax(scores, axis=1))
    249         y_pred = np.hstack(y_pred)
    250         acc = np.mean(y_pred == y)

```

```

    <__array_function__ internals> in argmax(*args, **kwargs)

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
-> argmax(a, axis, out)
    1186
    1187     """
-> 1188     return _wrapfunc(a, 'argmax', axis=axis, out=out)
    1189
    1190

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
-> _wrapfunc(obj, method, *args, **kwds)
    53     bound = getattr(obj, method, None)
    54     if bound is None:
---> 55         return _wrapit(obj, method, *args, **kwds)
    56
    57     try:

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
-> _wrapit(obj, method, *args, **kwds)
    42     except AttributeError:
    43         wrap = None
---> 44     result = getattr(asarray(obj), method)(*args, **kwds)
    45     if wrap:
    46         if not isinstance(result, mu.ndarray):

```

AxisError: axis 1 is out of bounds for array of dimension 1

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

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

```

num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

```

```

##### TODO: find the best lr and wight scale #####
learning_rate = 2e-3 # Experiment with this!
weight_scale = 9e-2 # Experiment with this!
#####
model = FullyConnectedNet([100, 100, 100, 100],
                           weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                 print_every=10, num_epochs=20, batch_size=25,
                 update_rule='sgd',
                 optim_config={
                     'learning_rate': learning_rate,
                 })
solver.train()

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

```

(Iteration 1 / 40) loss: 0.000000

```

└─
└─
AxisError                                Traceback (most recent call
└─last)

<ipython-input-16-83d126b9ecc6> in <module>()
    23         }
    24     )
--> 25 solver.train()
    26
    27 plt.plot(solver.loss_history, 'o')

/content/drive/My Drive/computer vision/assignments/assignment2/
└─CV7062610/solver.py in train(self)
    284         if first_it or last_it or epoch_end:
    285             train_acc = self.check_accuracy(
--> 286                 self.X_train, self.y_train, num_samples=self.
└─num_train_samples
    287             )
    288             val_acc = self.check_accuracy(

```

```

/content/drive/My Drive/computer vision/assignments/assignment2/
CV7062610/solver.py in check_accuracy(self, X, y, num_samples, batch_size)
    246         end = (i + 1) * batch_size
    247         scores = self.model.loss(X[start:end])
--> 248         y_pred.append(np.argmax(scores, axis=1))
    249         y_pred = np.hstack(y_pred)
    250         acc = np.mean(y_pred == y)

```

```

<__array_function__ internals> in argmax(*args, **kwargs)

```

```

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
argmax(a, axis, out)
    1186
    1187     """
-> 1188     return _wrapfunc(a, 'argmax', axis=axis, out=out)
    1189
    1190

```

```

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
_wrapfunc(obj, method, *args, **kwds)
    53     bound = getattr(obj, method, None)
    54     if bound is None:
---> 55         return _wrapit(obj, method, *args, **kwds)
    56
    57     try:

```

```

/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py in
_wrapit(obj, method, *args, **kwds)
    42     except AttributeError:
    43         wrap = None
---> 44     result = getattr(asarray(obj), method)(*args, **kwds)
    45     if wrap:
    46         if not isinstance(result, mu.ndarray):

```

```

AxisError: axis 1 is out of bounds for array of dimension 1

```

10.2 Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive

to the initialization scale? Why do you think that is the case?

10.3 Answer:

[FILL THIS IN]

11 Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 40% accuracy on the validation set using a fully-connected net. Using `sgd` this shouldn't take more than 5 minutes per training over 10 epochs.

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

```
[17]: best_model = None
best_val = 0.0
#####
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might
→#
# find batch/layer normalization and dropout useful. Store your best model in
→#
# the best_model variable.
→#
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

12 Test your model!

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

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

```

└─
-----
AttributeError                                Traceback (most recent call└─
└─last)

<ipython-input-18-bc662ae98aae> in <module>()
----> 1 y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
      2 y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
      3 print('Validation set accuracy: ', (y_val_pred == data['y_val']).
└─mean())
      4 print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())

AttributeError: 'NoneType' object has no attribute 'loss'
```

Convolutional Networks

April 25, 2021

```
[1]: # this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'CV7062610/assignments/assignment3/'
FOLDERNAME = 'computer vision/assignments/assignment2'
assert FOLDERNAME is not None, "[!] Enter the foldername."

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

# this downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/My\ Drive/$FOLDERNAME/CV7062610/datasets/
!bash get_datasets.sh
%cd /content
```

```
Mounted at /content/drive
/content/drive/My Drive/computer
vision/assignments/assignment2/CV7062610/datasets
/content
```

1 Convolutional Networks

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

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

```
[2]: # As usual, a bit of setup
import numpy as np
```

```

import matplotlib.pyplot as plt
from CV7062610.classifiers.cnn import *
from CV7062610.data_utils import get_CIFAR10_data
from CV7062610.gradient_check import eval_numerical_gradient_array,
    ↪eval_numerical_gradient
from CV7062610.layers import *
from CV7062610.fast_layers import *
from CV7062610.solver import Solver

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

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

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

```

[3]: # Load the (preprocessed) CIFAR10 data.

```

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

```

```

X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)

```

2 Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file CV7062610/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:

```
[4]: x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

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

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

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

3 Aside: Image processing via convolutions

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

3.1 Colab Users Only

Please execute the below cell to copy two cat images to the Colab VM.

```
[5]: # Colab users only!
%mkdir -p CV7062610/notebook_images
%cd drive/My\ Drive/$FOLDERNAME/CV7062610
%cp -r notebook_images/ /content/CV7062610/
%cd /content/
```

```
/content/drive/My Drive/computer vision/assignments/assignment2/CV7062610
/content
```

```

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

      kitten = imread('CV7062610/notebook_images/kitten.jpg')
      puppy = imread('CV7062610/notebook_images/puppy.jpg')
      # kitten is wide, and puppy is already square
      d = kitten.shape[1] - kitten.shape[0]
      kitten_cropped = kitten[:, d//2:-d//2, :]

      img_size = 200 # Make this smaller if it runs too slow
      resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
      resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size,
      →img_size)))

      x = np.zeros((2, 3, img_size, img_size))
      x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
      x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))

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

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

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

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

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

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

      # Show the original images and the results of the conv operation

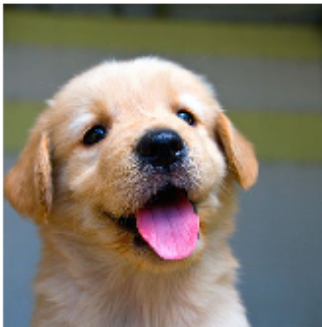
```

```

plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()

```

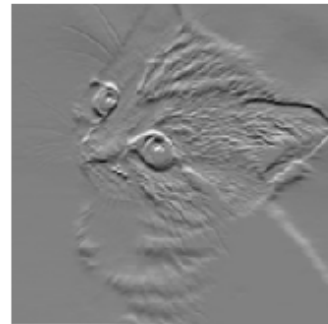
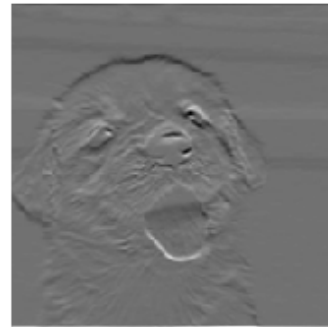
Original image



Grayscale



Edges



4 Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `CV7062610/layers.py`. Again, you don't need to worry too much about computational efficiency.

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

```
[ ]: np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

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

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

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

5 Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `CV7062610/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

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

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                           [-0.20421053, -0.18947368]],
                          [[-0.14526316, -0.13052632],
                           [-0.08631579, -0.07157895]]],
```



```

[[[-0.02736842, -0.01263158],
  [ 0.03157895,  0.04631579]]],
[[[ 0.09052632,  0.10526316],
  [ 0.14947368,  0.16421053]],
[[ 0.20842105,  0.22315789],
  [ 0.26736842,  0.28210526]],
[[ 0.32631579,  0.34105263],
  [ 0.38526316,  0.4         ]]]])

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

```

Testing max_pool_forward_naive function:
difference: 4.1666665157267834e-08

6 Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `CV7062610/layers.py`. You don't need to worry about computational efficiency.

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

```

[9]: np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

```

Testing max_pool_backward_naive function:
dx error: 3.27562514223145e-12

7 Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `CV7062610/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it either execute the local development cell (option A) if you are developing locally, or the Colab cell (option B) if you are running this assignment in Colab.

Very Important, Please Read. For **both** option A and B, you have to **restart** the notebook after compiling the cython extension. In Colab, please save the notebook File -> Save, then click Runtime -> Restart Runtime -> Yes. This will restart the kernel which means local variables will be lost. Just re-execute the cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

7.1 Option A: Colab

Execute the cell below only only **ONCE**.

```
[68]: %cd drive/My\ Drive/$FOLDERNAME/CV7062610/  
      !python setup.py build_ext --inplace
```

```
/content/drive/My Drive/computer vision/assignments/assignment2/CV7062610  
Compiling im2col_cython.pyx because it changed.  
[1/1] Cythonizing im2col_cython.pyx  
/usr/local/lib/python3.7/dist-packages/Cython/Compiler/Main.py:369:  
FutureWarning: Cython directive 'language_level' not set, using 2 for now (Py2).  
This will change in a later release! File: /content/drive/My Drive/computer  
vision/assignments/assignment2/CV7062610/im2col_cython.pyx  
    tree = Parsing.p_module(s, pxd, full_module_name)  
running build_ext  
building 'im2col_cython' extension  
creating build  
creating build/temp.linux-x86_64-3.7  
x86_64-linux-gnu-gcc -pthread -Wno-unused-result -Wsign-compare -DNDEBUG -g  
-fwrapv -O2 -Wall -g -fdebug-prefix-  
map=/build/python3.7-a56wZI/python3.7-3.7.10=. -fstack-protector-strong -Wformat  
-Werror=format-security -g -fdebug-prefix-  
map=/build/python3.7-a56wZI/python3.7-3.7.10=. -fstack-protector-strong -Wformat  
-Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -fPIC  
-I/usr/local/lib/python3.7/dist-packages/numpy/core/include  
-I/usr/include/python3.7m -c im2col_cython.c -o build/temp.linux-  
x86_64-3.7/im2col_cython.o  
In file included from /usr/local/lib/python3.7/dist-  
packages/numpy/core/include/numpy/ndarraytypes.h:1822:0,  
                 from /usr/local/lib/python3.7/dist-  
packages/numpy/core/include/numpy/ndarrayobject.h:12,  
                 from /usr/local/lib/python3.7/dist-  
packages/numpy/core/include/numpy/arrayobject.h:4,  
                 from im2col_cython.c:629:
```

```

/usr/local/lib/python3.7/dist-
packages/numpy/core/include/numpy/np_1_7_deprecated_api.h:17:2:
warning: #warning "Using deprecated NumPy API, disable it with
" "#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-Wcpp]
#warning "Using deprecated NumPy API, disable it with " \
~~~~~
x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl
,-Bsymbolic-functions -Wl,-z,relro -Wl,-Bsymbolic-functions -Wl,-z,relro -g
-fdebug-prefix-map=/build/python3.7-a56wZI/python3.7-3.7.10=. -fstack-protector-
strong -Wformat -Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2
build/temp.linux-x86_64-3.7/im2col_cython.o -o /content/drive/My Drive/computer
vision/assignments/assignment2/CV7062610/im2col_cython.cpython-37m-x86_64-linux-
gnu.so

```

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

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

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

```

[ ]: # Rel errors should be around e-9 or less
from CV7062610.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

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

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

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

```

```

t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

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

```

```

[11]: # Relative errors should be close to 0.0
from CV7062610.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

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

```

```

Testing pool_forward_fast:
Naive: 0.202086s
fast: 0.003035s
speedup: 66.583582x
difference: 0.0

```

```
Testing pool_backward_fast:
Naive: 1.419938s
fast: 0.012973s
speedup: 109.454753x
dx difference: 0.0
```

8 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file CV7062610/layer_utils.py you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check they're working.

```
[12]: from CV7062610.layer_utils import conv_relu_pool_forward, \
      ↪conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

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

```
Testing conv_relu_pool
dx error: 9.591132621921372e-09
dw error: 5.802391137330214e-09
db error: 1.0146343411762047e-09
```

```
[13]: from CV7062610.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
```

```

x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

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

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

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))

```

Testing conv_relu:

```

dx error:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10

```

9 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 `CV7062610/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the `fast/sandwich` layers (already imported for you) in your implementation. Run the following cells to help you debug:

9.1 Sanity check loss

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

```

[18]: model = ThreeLayerConvNet()

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

```

```

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

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

```

```

Initial loss (no regularization): 2.3025870957074215
Initial loss (with regularization): 2.5089342631993343

```

9.2 Gradient check

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

```

[19]: num_inputs = 2
      input_dim = (3, 16, 16)
      reg = 0.0
      num_classes = 10
      np.random.seed(231)
      X = np.random.randn(num_inputs, *input_dim)
      y = np.random.randint(num_classes, size=num_inputs)

      model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                                input_dim=input_dim, hidden_dim=7,
                                dtype=np.float64)

      loss, grads = model.loss(X, y)
      # Errors should be small, but correct implementations may have
      # relative errors up to the order of e-2
      for param_name in sorted(grads):
          f = lambda _: model.loss(X, y)[0]
          param_grad_num = eval_numerical_gradient(f, model.params[param_name],
      ↪ verbose=False, h=1e-6)
          e = rel_error(param_grad_num, grads[param_name])
          print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
      ↪ grads[param_name])))

```

```

W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10

```

9.3 Overfit small data

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

```
[21]: np.random.seed(231)

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

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                 num_epochs=35, batch_size=50,
                 update_rule='sgd',
                 optim_config={
                     'learning_rate': 1e-3,
                 },
                 verbose=True, print_every=1)
solver.train()

(Iteration 1 / 70) loss: 2.414060
(Epoch 0 / 35) train acc: 0.140000; val_acc: 0.076000
(Iteration 2 / 70) loss: 2.388208
(Epoch 1 / 35) train acc: 0.130000; val_acc: 0.075000
(Iteration 3 / 70) loss: 2.286111
(Iteration 4 / 70) loss: 2.280937
(Epoch 2 / 35) train acc: 0.190000; val_acc: 0.095000
(Iteration 5 / 70) loss: 2.156452
(Iteration 6 / 70) loss: 2.121301
(Epoch 3 / 35) train acc: 0.220000; val_acc: 0.102000
(Iteration 7 / 70) loss: 2.082505
(Iteration 8 / 70) loss: 2.132747
(Epoch 4 / 35) train acc: 0.240000; val_acc: 0.113000
(Iteration 9 / 70) loss: 1.974502
(Iteration 10 / 70) loss: 2.234853
(Epoch 5 / 35) train acc: 0.270000; val_acc: 0.115000
(Iteration 11 / 70) loss: 2.062134
(Iteration 12 / 70) loss: 1.967815
(Epoch 6 / 35) train acc: 0.270000; val_acc: 0.115000
(Iteration 13 / 70) loss: 2.151717
(Iteration 14 / 70) loss: 1.962033
(Epoch 7 / 35) train acc: 0.330000; val_acc: 0.131000
```


(Iteration 15 / 70) loss: 1.905878
(Iteration 16 / 70) loss: 1.981501
(Epoch 8 / 35) train acc: 0.420000; val_acc: 0.144000
(Iteration 17 / 70) loss: 2.000987
(Iteration 18 / 70) loss: 1.943681
(Epoch 9 / 35) train acc: 0.420000; val_acc: 0.137000
(Iteration 19 / 70) loss: 1.894658
(Iteration 20 / 70) loss: 1.798240
(Epoch 10 / 35) train acc: 0.390000; val_acc: 0.137000
(Iteration 21 / 70) loss: 1.930483
(Iteration 22 / 70) loss: 1.848953
(Epoch 11 / 35) train acc: 0.460000; val_acc: 0.160000
(Iteration 23 / 70) loss: 1.733071
(Iteration 24 / 70) loss: 1.833933
(Epoch 12 / 35) train acc: 0.520000; val_acc: 0.167000
(Iteration 25 / 70) loss: 1.759339
(Iteration 26 / 70) loss: 1.707899
(Epoch 13 / 35) train acc: 0.530000; val_acc: 0.171000
(Iteration 27 / 70) loss: 1.668286
(Iteration 28 / 70) loss: 1.632742
(Epoch 14 / 35) train acc: 0.510000; val_acc: 0.173000
(Iteration 29 / 70) loss: 1.493473
(Iteration 30 / 70) loss: 1.852287
(Epoch 15 / 35) train acc: 0.590000; val_acc: 0.177000
(Iteration 31 / 70) loss: 1.544719
(Iteration 32 / 70) loss: 1.445474
(Epoch 16 / 35) train acc: 0.620000; val_acc: 0.167000
(Iteration 33 / 70) loss: 1.438357
(Iteration 34 / 70) loss: 1.558215
(Epoch 17 / 35) train acc: 0.550000; val_acc: 0.181000
(Iteration 35 / 70) loss: 1.457353
(Iteration 36 / 70) loss: 1.430685
(Epoch 18 / 35) train acc: 0.640000; val_acc: 0.177000
(Iteration 37 / 70) loss: 1.417343
(Iteration 38 / 70) loss: 1.224349
(Epoch 19 / 35) train acc: 0.560000; val_acc: 0.182000
(Iteration 39 / 70) loss: 1.358695
(Iteration 40 / 70) loss: 1.459283
(Epoch 20 / 35) train acc: 0.620000; val_acc: 0.171000
(Iteration 41 / 70) loss: 1.324597
(Iteration 42 / 70) loss: 1.141805
(Epoch 21 / 35) train acc: 0.710000; val_acc: 0.176000
(Iteration 43 / 70) loss: 1.251012
(Iteration 44 / 70) loss: 1.255930
(Epoch 22 / 35) train acc: 0.740000; val_acc: 0.183000
(Iteration 45 / 70) loss: 1.265682
(Iteration 46 / 70) loss: 1.241632
(Epoch 23 / 35) train acc: 0.800000; val_acc: 0.182000

```
(Iteration 47 / 70) loss: 1.088461
(Iteration 48 / 70) loss: 1.286929
(Epoch 24 / 35) train acc: 0.790000; val_acc: 0.187000
(Iteration 49 / 70) loss: 1.043943
(Iteration 50 / 70) loss: 1.232451
(Epoch 25 / 35) train acc: 0.830000; val_acc: 0.186000
(Iteration 51 / 70) loss: 1.166305
(Iteration 52 / 70) loss: 0.992921
(Epoch 26 / 35) train acc: 0.810000; val_acc: 0.183000
(Iteration 53 / 70) loss: 1.027397
(Iteration 54 / 70) loss: 1.034611
(Epoch 27 / 35) train acc: 0.870000; val_acc: 0.191000
(Iteration 55 / 70) loss: 1.074365
(Iteration 56 / 70) loss: 0.884611
(Epoch 28 / 35) train acc: 0.790000; val_acc: 0.180000
(Iteration 57 / 70) loss: 1.045844
(Iteration 58 / 70) loss: 0.771730
(Epoch 29 / 35) train acc: 0.850000; val_acc: 0.188000
(Iteration 59 / 70) loss: 0.964063
(Iteration 60 / 70) loss: 0.919592
(Epoch 30 / 35) train acc: 0.840000; val_acc: 0.190000
(Iteration 61 / 70) loss: 0.898468
(Iteration 62 / 70) loss: 0.837345
(Epoch 31 / 35) train acc: 0.890000; val_acc: 0.194000
(Iteration 63 / 70) loss: 0.866577
(Iteration 64 / 70) loss: 0.730397
(Epoch 32 / 35) train acc: 0.850000; val_acc: 0.188000
(Iteration 65 / 70) loss: 0.562069
(Iteration 66 / 70) loss: 0.911381
(Epoch 33 / 35) train acc: 0.910000; val_acc: 0.179000
(Iteration 67 / 70) loss: 0.662602
(Iteration 68 / 70) loss: 0.702882
(Epoch 34 / 35) train acc: 0.920000; val_acc: 0.182000
(Iteration 69 / 70) loss: 0.514115
(Iteration 70 / 70) loss: 0.786051
(Epoch 35 / 35) train acc: 0.900000; val_acc: 0.200000
```

10 i changed the update_rule to sgd because that is the only optimizer we have.

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

Small data training accuracy: 0.9

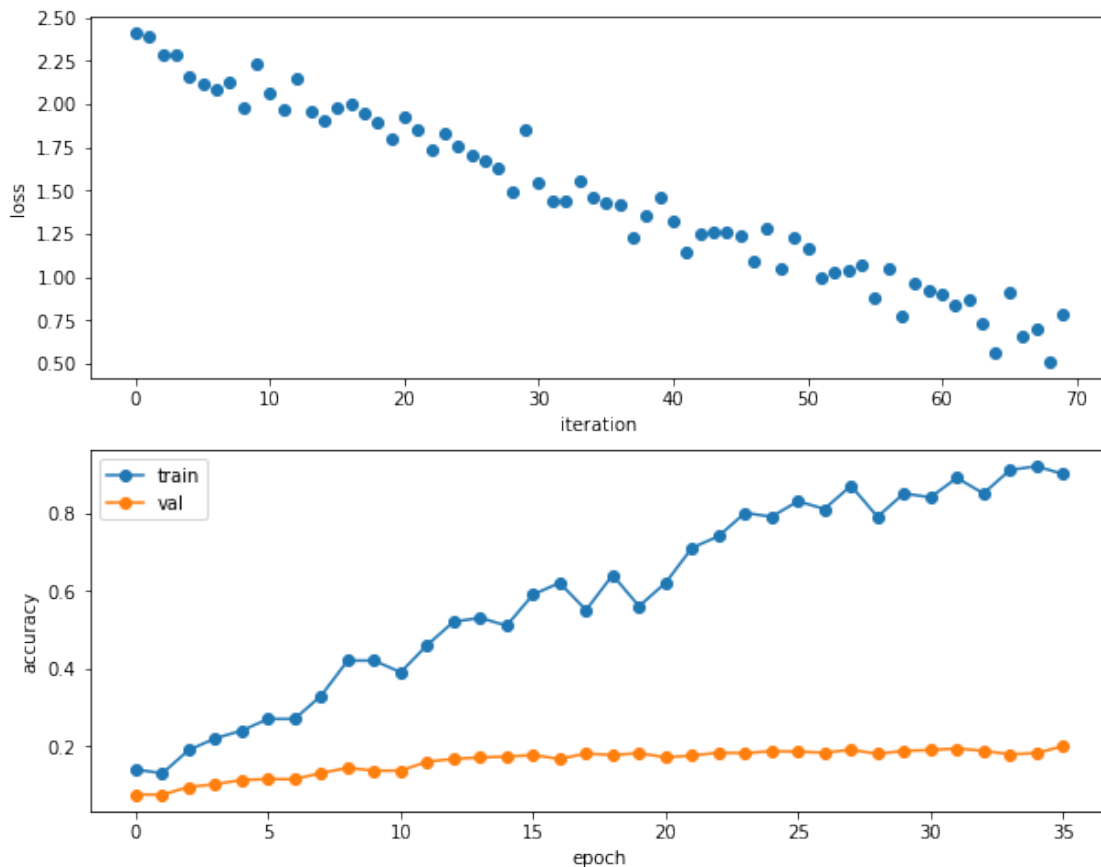
```
[23]: # Print final validation accuracy
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

Small data validation accuracy: 0.2

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

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

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



10.1 Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 30% accuracy on the training set with sgd, it should take 6 minuts. In the next part when we will learn other optimization tequincs we will get better accuracy in the same amount of time:

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

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

```
(Iteration 1 / 980) loss: 2.304876
(Epoch 0 / 1) train acc: 0.083000; val_acc: 0.086000
(Iteration 21 / 980) loss: 2.304521
(Iteration 41 / 980) loss: 2.304424
(Iteration 61 / 980) loss: 2.304157
(Iteration 81 / 980) loss: 2.304181
(Iteration 101 / 980) loss: 2.303648
(Iteration 121 / 980) loss: 2.302706
(Iteration 141 / 980) loss: 2.303352
(Iteration 161 / 980) loss: 2.301611
(Iteration 181 / 980) loss: 2.298818
(Iteration 201 / 980) loss: 2.293071
(Iteration 221 / 980) loss: 2.285592
(Iteration 241 / 980) loss: 2.272385
(Iteration 261 / 980) loss: 2.244782
(Iteration 281 / 980) loss: 2.208467
(Iteration 301 / 980) loss: 2.050966
(Iteration 321 / 980) loss: 2.058296
(Iteration 341 / 980) loss: 2.258100
(Iteration 361 / 980) loss: 2.037670
(Iteration 381 / 980) loss: 1.991332
(Iteration 401 / 980) loss: 1.982192
(Iteration 421 / 980) loss: 1.943203
(Iteration 441 / 980) loss: 2.018387
(Iteration 461 / 980) loss: 2.059734
(Iteration 481 / 980) loss: 1.910822
(Iteration 501 / 980) loss: 1.942802
(Iteration 521 / 980) loss: 1.786348
```

```
(Iteration 541 / 980) loss: 2.038088
(Iteration 561 / 980) loss: 1.912679
(Iteration 581 / 980) loss: 1.921920
(Iteration 601 / 980) loss: 1.987053
(Iteration 621 / 980) loss: 1.979135
(Iteration 641 / 980) loss: 1.866771
(Iteration 661 / 980) loss: 1.710859
(Iteration 681 / 980) loss: 2.107062
(Iteration 701 / 980) loss: 1.754281
(Iteration 721 / 980) loss: 1.917319
(Iteration 741 / 980) loss: 2.004374
(Iteration 761 / 980) loss: 1.546441
(Iteration 781 / 980) loss: 1.817538
(Iteration 801 / 980) loss: 1.697297
(Iteration 821 / 980) loss: 1.391700
(Iteration 841 / 980) loss: 1.776255
(Iteration 861 / 980) loss: 1.777225
(Iteration 881 / 980) loss: 1.706812
(Iteration 901 / 980) loss: 1.600886
(Iteration 921 / 980) loss: 1.496853
(Iteration 941 / 980) loss: 1.674209
(Iteration 961 / 980) loss: 1.514064
(Epoch 1 / 1) train acc: 0.402000; val_acc: 0.417000
```

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

Full data training accuracy: 0.34

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

Full data validation accuracy: 0.417

10.2 Visualize Filters

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

```
[28]: from CV7062610.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
```

```
plt.imshow(grid.astype('uint8'))  
plt.axis('off')  
plt.gcf().set_size_inches(5, 5)  
plt.show()
```

