CSC321 Winter 2015 - Assignment 2
Convolutional Neural Nets

Due Date: Tuesday March 10, 2015 (at the start of the class).
TA: Ryan Kiros (`csc321ta@cs.toronto.edu`)

In this assignment we apply neural networks on images. We will be working with the USPS dataset of digit images. We will train some fully connected networks as well as convolutional networks. There are two main goals of this assignment.

- Understand how to construct neural networks in a modular way. The starter code is designed in a way which should encourage you to think of layers as modules, forward and backpropagation as local operations in these modules, and a full neural network as just a concatenation of these local modules.
- Understand convolutions. In the class, we saw two ways of interpreting convolutions. In this assignment, you see implementations of both.

Part 0: Getting set up

Download the starter code. This includes the data as well.

Dataset: The USPS dataset consists of 16×16 handwritten digit images extracted from postal codes people wrote on letters. There are 10 class labels corresponding to the digits 0, . . ., 9. The training set has 7291 images from which we hold out 1000 random ones for validation. The test set has 2007 images.

Testing the convolutions: The starter code includes an implementation of a `convolve` method written in Cython (`convolutions.pyx`). It needs to be compiled in order to use it. However, if you are running this assignment on the CDF machines, you do not need to compile this because a precompiled shared library `convolutions.so` is included with the starter code. If you have a different setup, you will need to compile it. To do that, run

```
python setup.py build_ext --inplace
```

This will produce `convolutions.so` which can be imported into python. The compilation requires you to have Cython installed. It may give some warnings if your Numpy is old, but you can ignore them. Run the `play_with_convolutions.py` script.

```
python play_with_convolutions.py
```

This script loads an image and convolves it with three filters which we saw in class. If this script runs fine and shows you the output, then you are all set to use the `convolve` method.

Specifying network architectures: In this assignment we will be training many different network architectures. To make it easy to specify these architectures, we will use configuration files written in YAML (a fairly easy to use markup language). Check out some of the YAML files. They specify the type of layers, number of hidden units and convolutional parameters. They also include hyperparameters such as the initial weight scale, learning rate, momentum and l2 decay. Additionally, they also specifies where the data file is located and the name of the checkpoint file that the model will output. The checkpoint file contains the learned model parameters. The config file also specifies the number of epochs to run and batch size to use.

Part 1: Fully connected nets

In this part, you will complete the implementation of `FCLayer.ComputeUp` and `FCLayer.ComputeDown` methods in `fc_layer.py` and run some experiments using fully connected nets. The implementations can be done in a total of 4-5 lines of code, although you are free to use more. DO NOT write loops.

1. [2 points] Implement `FCLayer.ComputeUp`. This method forward propagates activations.

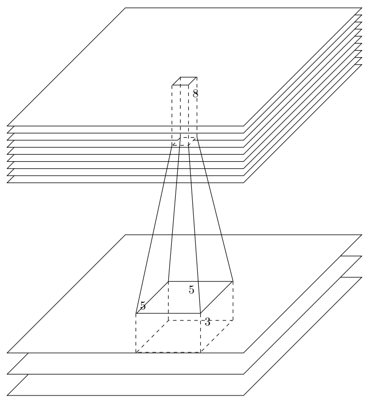
2. [4 points] Implement `FClayer.ComputeDown`. This method backprops gradients.
3. Run the gradient checker to make sure your implementation is correct.
   `python grad_check.py logreg.yaml`
   If it says PASSED then you are ready to proceed.
4. At this point you will be able to train fully connected networks. We will first start off with logistic regression. The model is described in `logreg.yaml`. This is essentially a neural net with no hidden layers. You can train this model by running
   `python train.py logreg.yaml`
   As the model trains it will output the performance metrics evaluated after each epoch. After every few updates, it will also show a random training example and the activations of all the layers for that training example. After the model has finished training, the checkpoint file `logreg.npz` will contain the model that achieved the best validation performance. To evaluate the final model, run
   `python evaluate.py logreg.yaml`
   This will load the checkpoint file and run the model on the training, validation and test sets. Record these numbers.
5. Similarly, train and evaluate `net_1layer.yaml`. This is a one hidden layer neural net. Record the numbers for the three sets.
6. Train and evaluate `net_2layer.yaml`. This is a two hidden layer neural net. Record the numbers for the three sets.
7. [3 points] In your write up, put the recorded numbers into a table. This table should have three rows and six columns. The rows correspond to the three models you trained. The columns correspond to the six numbers : Avg cross entropy and classification accuracy for each of the three splits. 1 point for each row.
8. [1 point] Now we can analyze these results and compare the three models. Which model would you consider the best and why ? Which one gets the highest training accuracy ? Which one gets the highest test accuracy ?
9. [Extra; 0 points] If you want to play more with fully connected nets, try changing the sizes of the hidden layers, learning rates, momentum, may be add more layers and see which models work better than others.

Part 2: Convolutional nets

In this part, you will complete the implementation of `ConvLayer.ComputeUp` and `ConvLayer.ComputeDown` methods in `conv_layer.py` and run some experiments using convolutional nets.

Channels, kernels, strides and padding We will be using these terms when talking about convolutional network architectures. Here we describe these terms so that the network architectures can be understood very clearly.

Suppose we have an image of spatial size 40 × 40 that consists of 3 *channels*. This means the image has a width and height of 40 pixels, and at each pixel location, we have 3 numbers. For color images, the three channels often correspond to RGB (red, green and blue) intensities. A channel is also called a *feature map* - a *map* because it has spatial size, and *feature* because it represents some feature of the input. For example, the first feature map represents the redness of the image, the second represents the greenness etc. More generally, any hidden layer of a convolutional network will have some spatial size and some number of channels. Therefore, any layer can be thought of as a cuboid having some width, height and number of channels. For hidden layers, unlike the input, we don't know what the channels represent. Their meaning will be learned. But we need to pick the *number* of channels, just like we picked the number of hidden units for fully connected layers.



A **kernel** (or synonymously, a filter) is the set of weights that are "applied" on an input patch. A patch will have spatial size and number of channels. For example, in the context of the input image described before, we can talk about a 5 × 5 patch cutting across all three channels of the input. Such a patch will have 75 numbers. Therefore, the weights/kernel/filter connected to this patch will have 75 parameters. "Applying" the kernel on this patch means computing the dot product of the two 75-d vectors. Applying the kernel *convolutionally* means applying the same kernel to different patches of the input. By applying the kernel at different (overlapping) patches, we get one *output map* or *output channel*. In general, we can apply k different filters convolutionally and obtain k output maps. The term *kernel* or *convolutional weights* often refers to this collection of multiple filters. Therefore, we can say something like, "a kernel of spatial size 5 × 5 having 3 input channels and 8 output channels". This kernel would have $5 * 5 * 3 * 8$ parameters. It can be applied convolutionally on an input of 3 channels and any spatial size. It will produce an output of 8 channels and some spatial size.

Strides: When we apply a kernel convolutionally, we would like to have some control over where the kernel is applied. The default way of applying convolutions is to apply the filter at *all* locations in the input. For computational or other reasons, we may not want to apply a filter at all locations but skip some locations. The term **stride** describes after how many locations should the kernel be applied again. A stride of 1, means that the kernel will be applied at all locations. A stride of 2, means that we skip 1 location, etc.

Padding: Boundary issues come up when we try to apply a kernel near the borders of the input. For example, if we have a 5×5 kernel, and we insist on having the kernel look at only real pixels, the first place where we can apply the kernel is when its center rests on pixel $(2, 2)$ of the input. We cannot apply it when its center rests on $(0, 0)$ because then part of the kernel will have no image pixels feeding in. This means that pixels near the boundary of the image will receive much less attention. It is probably OK to ignore this at the input layer for a big image. But this becomes a problem as we go higher in a convolutional net and input sizes become small. For example, if we have a 13th hidden layer of spatial size 4×4 and we want to apply a 3×3 kernel, we only have 4 locations to apply it on out of the 16 present. In order to salvage something from this bad situation, we can implicitly pad the input layer with a border of zeros along all four sides. The width of this border is the *padding*. So if we choose a padding of 1, the input implicitly becomes 6×6 and now, we can apply the filter on 16 positions.

So now you understand what it means to "apply a 5×5 kernel with 8 input channels and 32 output channels, convolutionally with a stride of 2 and padding of 2".

The same terminology applies to pooling layers as well.

Some warm-up questions : You don't have to submit the answers in the writeup.

Suppose the input to a convolution layer has a spatial size 16×16 with 8 channels. We want to convolve this input with a 5×5 kernel with 32 output channels.

1. What is the spatial size of the output if the convolutions are applied with a stride of 1 and no padding ?
2. How many total units do we have in the output layer ? The output layer can be thought of as a cuboid with the three dimensions being spatial size along X (width), spatial size along Y (height) and the number of channels.
3. How many parameters are there in the convolutional weights ? How many in the bias ? There is one bias parameter per output channel.
4. How many weight parameters would there be if we wanted to have an output layer of the same size as computed above, but which was fully connected to the input layer (that is, each output unit was connected to each input unit) ?

5. How much padding should we use if we want the spatial size of the output to be the same as that of the input. (Assuming a stride of 1) ?

6. If instead of applying the filters at each location (stride=1), we have a stride of 2, what would be the spatial size of the output layer (assume no padding) ?

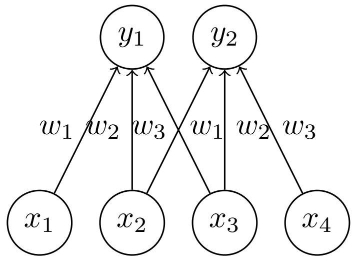
7. Suppose we have an input of size $i_y \times i_x$. We convolve it with a kernel of size $k \times k$ with a stride of s and a padding of p. Derive an expression for the size of the output layer in terms of i_y, i_x, k, s and p.

Implementing convolutions
In class, we mentioned two ways to think about convolutions.

- Method 1: Apply a filter at different regions of a signal by taking the dot product of the filter with that region.
- Method 2: Go to each position in the signal and multiply the signal value into all the locations of the filter. Translate the multiplied filters according to the position and add them up.

In the starter code, the first method is already implemented. You will be implementing the second method using the given `convolve` method. Please see the comment describing this function in `convolutions.pyx` to understand how to use it. You do not need to understand the details of how it works, just think of it as a black box convolution method. Take a look at `play_with_convolutions.py` to see an example of using this method.

In order to understand the two methods better, consider this 1-D example. Suppose we have an input of length 4 : (x_1, x_2, x_3, x_4). We want to convolve it with a kernel of length 3: (w_1, w_2, w_3). The network is shown alongside. We are using a stride of 1 and padding of 0. We will now do forward prop and backprop on this network using both methods.



Forward prop
Method 1: Take the dot product of the filter with (x_1, x_2, x_3) to get $y_1 = w_1x_1 + w_2x_2 + w_3x_3$.
Then take the dot product of the filter with (x_2, x_3, x_4) to get $y_2 = w_1x_2 + w_2x_3 + w_3x_4$.

Method 2: Flip the filter, so we get (w_3, w_2, w_1). Convolve (x_1, x_2, x_3, x_4) with (w_3, w_2, w_1).

Multiply by x_1	x_1w_3	x_1w_2	x_1w_1			
Multiply by x_2		x_2w_3	x_2w_2	x_2w_1		
Multiply by x_3			x_3w_3	x_3w_2	x_3w_1	
Multiply by x_4				x_4w_3	x_4w_2	x_4w_1
Add	x_1w_3	$x_1w_2 + x_2w_3$	$x_1w_1 + x_2w_2 + x_3w_3$	$x_2w_1 + x_3w_2 + x_4w_3$	$x_3w_1 + x_4w_2$	x_4w_1
	dropped	dropped	y_1	y_2	dropped	dropped

We get the same y_1 and y_2 as in Method 1. Note that terms that did not involve all the weights were dropped. The `convolve` method does the same, by default. In order to retain some of these terms, we would have to use padding. For example, a padding of length 1 will implicitly add one zero on either side of the input. Therefore, one extra term will be retained on either side.

Backprop (Input derivatives)
Now suppose we have derivatives $(\frac{\partial C}{\partial y_1}, \frac{\partial C}{\partial y_2})$. Lets call them (d_1, d_2). We want to compute the derivatives w.r.t the inputs. i.e., $(\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \frac{\partial C}{\partial x_3}, \frac{\partial C}{\partial x_4})$
Method 1:
x_1 is connected to only y_1. So it gets the derivative $\frac{\partial C}{\partial x_1} = w_1d_1$
x_2 is connected to both y_1, y_2. So it gets the derivative $\frac{\partial C}{\partial x_2} = w_2d_1 + w_1d_2$
x_3 is connected to both y_1, y_2. So it gets the derivative $\frac{\partial C}{\partial x_3} = w_3d_1 + w_2d_2$

x_4 is connected to only y_2. So it gets the derivative $\frac{\partial C}{\partial x_4} = w_3 d_2$

Method 2:
Use the unflipped weights. Convolve (d_1, d_2) with (w_1, w_2, w_3).

Multiply by d_1	d_1w_1	d_1w_2	d_1w_3	
Multiply by d_2		d_2w_1	d_2w_2	d_2w_3
Add	d_1w_1	$d_1w_2 + d_2w_1$	$d_1w_3 + d_2w_2$	d_2w_3
	$\frac{\partial C}{\partial x_1}$	$\frac{\partial C}{\partial x_2}$	$\frac{\partial C}{\partial x_3}$	$\frac{\partial C}{\partial x_4}$

As we can see, using the convolution gives the same derivatives as got in method 1. Note that we need to retain all the terms. This corresponds to using a padding of length 2. Recall, that in the forward pass we used a padding of 0. In general, if we use a padding of p in the forward pass, we need a padding of $k - p - 1$ in the backward pass, where k is the kernel size.

Backprop (Weight derivatives)

Here again suppose we have derivatives $(\frac{\partial C}{\partial y_1}, \frac{\partial C}{\partial y_2})$. Lets call them (d_1, d_2). We want to compute the derivatives w.r.t the weights. i.e., $(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial w_3})$.

Method 1:
Weight w_1 connects (x_1, y_1) and (x_2, y_2). Therefore, $\frac{\partial C}{\partial w_1} = x_1d_1 + x_2d_2$.
Weight w_2 connects (x_2, y_1) and (x_3, y_2). Therefore, $\frac{\partial C}{\partial w_2} = x_2d_1 + x_3d_2$.
Weight w_3 connects (x_3, y_1) and (x_4, y_2). Therefore, $\frac{\partial C}{\partial w_3} = x_3d_1 + x_4d_2$.

Method 2:
Flip the derivatives to get (d_2, d_1). Convolve the inputs (x_1, x_2, x_3, x_4) with the flipped derivatives (d_2, d_1).

Multiply by x_1	x_1d_2	x_1d_1			
Multiply by x_2		x_2d_2	x_2d_1		
Multiply by x_3			x_3d_2	x_3d_1	
Multiply by x_4				x_4d_2	x_4d_1
Add	x_1d_2	$x_1d_1 + x_2d_2$	$x_2d_1 + x_3d_2$	$x_3d_1 + x_4d_2$	x_4d_1
	dropped	$\frac{\partial C}{\partial w_1}$	$\frac{\partial C}{\partial w_2}$	$\frac{\partial C}{\partial w_3}$	dropped

We can see that by convolving the inputs with the flipped derivatives, we get the derivatives w.r.t the weights. Here we dropped any terms that did not involve both derivatives. This corresponds to using a padding of 0. In general, we would use the same padding as the forward prop.

To summarize:

- Fprop : convolve(inputs, flipped weights, padding=p)
- Bprop (to get derivatives w.r.t inputs) : convolve(derivatives, weights, padding=k-p-1)
- Bprop (to get derivatives w.r.t weights) : convolve(inputs, flipped derivatives, padding=p)

Once you understand this, implementing the 2D version will be straightforward.

Now we can get into experiments and implementation:

1. We can start training convolutional networks using method 1 which is already implemented. Train and evaluate the `convnet1_method1.yaml` model.
2. [9 points]Implement Method 2 in `ConvLayer.ComputeUp` and `ConvLayer.ComputeDown`. This can be done in a total of 3 lines of code, although you should feel free to use more. DO NOT write loops.
3. Run gradient check on `convnet_gradcheck.yaml`. This runs gradient check using your implementation of Method 2. If this says PASSED your implementation is correct and you are ready to proceed. For grad check, we are using a very small model for which gradient check works reliably. For bigger models, gradient check may not be very accurate.

4. Train and evaluate the `convnet1_method2.yaml` model. This will train the `convnet1` architecture using your implementation of Method 2. This should produce the exact same run as Method 1 because the same computation is being performed in a different way.

5. Train and evaluate the `convnet2.yaml` model. This network is the same as the first one, except that it uses 5×5 filters instead of 3×3.

6. [2 points] Report the results for the two models (2 rows, 6 numbers each).

7. [1 points] Compare the two convolutional models and the three previously trained fully connected models. Which one does better ? convolutional or fully connected ? Also mention how you decided which model is better.

8. [6 points] The `convnet2.yaml` model has 6 layers - `conv_1`, `maxpool_1`, `conv_2` , `maxpool_2`, `fc_1` and `output`. For each layer, calculate

   - The spatial size, number of channels and total number of hidden units. Fully connected layers have a spatial size of 1×1.
   - The number of connections. An output unit is "connected" to an input unit if the value of the output unit depends on that input unit. You have to count the number of connections.
   - The number of learnable parameters (For simplicity, ignore the biases).

   Also, briefly explain how you computed these numbers.

9. [2 points] Compute the total number of learnable parameters in the best fully connected model and the best convolutional model. One easy way to do this is to open the generated checkpoint files and look at the shapes of the weight and bias matrices.

10. [Extra; 0 points] Train the `convnet3.yaml` model. This is a deeper model and will take quite a lot of time to train. Also try playing with different architectures and hyperparameters. If you are looking for more challenging datasets, you can work with the MNIST, NORB, CIFAR-10/100 or SVHN datasets.

What you have to submit

For reference, here is everything you need to hand in:

- A printout of what you wrote in the two methods `FCLayer.ComputeUp` and `FCLayer.ComputeDown`. Please do not include any other code.

- A printout of what you wrote in the two methods `ConvLayer.ComputeUp` and `ConvLayer.ComputeDown`. Please do not include any other code.

- The analysis of the results from part 1, including the table of numbers.

- The answers to questions and the analysis of the results from part 2, including the table of numbers.

- Optional: Which part of this assignment did you find the most valuable? The most difficult and/or frustrating?

You may submit **at most 2 pages** for the analysis, although 1 page is sufficient and recommended. The other printouts do not count towards the page limit.

This assignment is graded out of 30 points: 10 for Part 1, 20 for Part 2.