CMPEN/EE 454, Project 1, Fall 2015

Due Weds Sep 23, 11:55PM, submitted via Angel

1 Motivation

The goal of this project is to implement the forward pass of a convolutional neural net (CNN) for object recognition in Matlab. One of the eye-opening developments in computer vision in recent few years is that a cascade or chain of very simple image processing operations can outperform more complicated state-of-the-art object recognition algorithms. Of course, the specific image processing operations used have to be well-chosen! The real magic in a CNN is learning the underlying convolution filters, using a process called backpropagation (we'll call this the backward pass). Much of the heavy computational and big-data requirements for training CNNs (so-called deep learning) are concentrated on this learning process, which is outside the scope of this course and project. However, once learned, applying a CNN in the forward direction to map an input image through a set of intermediate image processing transformations and finally to a set of object class probabilities involves little more than convolution with linear filters, image size reduction, and thresholding. This chain of operations is what we are going to implement in this project.

Specific project goals include:

- Gaining experience in Matlab programming for implementing vision algorithms
- Use of multi-dimensional arrays in Matlab
- Applying linear filters to an image with convolution
- Other image processing techniques: normalization, subsampling, thresholding
- Quantitative performance evaluation of a vision algorithm

2 The Basic Operations

A CNN operates in a set of stages, or layers, each of which produces an intermediate result. Input to each stage is an $N_1 \times M_1 \times D_1$ array, and output is an $N_2 \times M_2 \times D_2$ array. Input to the first stage is a color image (e.g. array of size 32x32x3); output from the final stage is a vector of probabilities, one per object class (e.g. array of size 1x1x10 if we have 10 different object classes), with higher probabilities meaning it is likely that the picture contains that kind of object. We are going to use computer vision nomenclature and think of an $N \times M \times D$ array as a multi-channel image having N rows, M columns, and D channels. *Channels* is just a word to describe the number of values associated with each 2D spatial pixel location in an image, so, for example, a greyscale image has 1 channel and a color image has 3 channels (red, green, blue). An image with D channels might be used to represent a multispectral image from a sensor that measures D wavelength ranges. In our case, D channels just means that you need D numbers to fully describe the value stored at each spatial (row, column) pixel location.

Different CNNs in the literature are made up of various numbers of layers, operating on images of various sizes, using various combinations/orderings of a basic library of computations. Our simple CNN is going to be built from the following computational building blocks.

Image Normalization: Input is a color image array of size $N \times M \times 3$ and output is an array of the same size. Each value in the output array is defined as

$$\text{Out}(i,j,k) = \text{In}(i,j,k)/255.0 - 0.5 \ ,$$

which approximately scales each color channel's pixel values into the output range -0.5 to 0.5.

ReLU: Input is an array of size $N \times M \times D$ and output is an array of the same size. Each value in the output array is defined as

$$\text{Out}(i,j,k) = \max(\text{In}(i,j,k) \ , \ 0) \ .$$

ReLU stands for Rectified Linear Unit, but despite the fancy name it is just a thresholding operation where any negative numbers in the input become 0 in the output.

Maxpool: Input is an array of size $2N \times 2M \times D$ and output is an array of the size $N \times M \times D$. If we think of the first two dimensions of an array as being rows and columns, then maxpool is reducing the spatial size of the array by producing an output having only half the number of rows and columns (so total number of elements in the array will be reduced by 4). Note that the number of channels, D, in the output array is the same as number of channels in the input array, so only the spatial dimensions are being reduced. Maxpool operates on each channel separately, and assigns output channel values by taking the maximum value in each nonoverlapping 2x2 block of the input channel. More precisely,

$$\text{Out}(i,j,k) = \max(\{ \ \text{In}(r,c,k) \mid (2i-1) < r < 2i \ \text{ and } (2j-1) < c < 2j \ \}) \ .$$

Figure 1 shows an example of reducing one channel of a 4x4 input image to get one channel of a 2x2 output image. You may assume the input image has an even number of rows and columns.

Convolution: Input is an array of size $N \times M \times D_1$ and output is an array of size $N \times M \times D_2$. Note that the spatial size of the image stays the same (number of rows and columns are the same), but the number of channels in the output image can differ from the input. The job of a convolution layer is to apply a set of predefined linear filters and bias values to the input image to compute each channel of the output image. This is the main computational process in a CNN, and the most difficult to implement.

To fully specify the linear transformation from input to output, we will need to know a set of D_2 scalar bias values and a set of D_2 filters of size $R \times C \times D_1$, where $R \times C$ is the spatial size of a filter, and typically is small compared to the $N \times M$ spatial size of the image. We will call the set of filters a *filter bank*, and represent it in Matlab as a 4-dimensional array of size $R \times C \times D_1 \times D_2$. That is, if F is our filter bank array, then $F(:,:,:,l)$ is the lth filter, $1 \leq l \leq D_2$, and that filter is a 3-dimensional array of size $R \times C \times D_1$. This lth filter is applied to the entire $N \times M \times D_1$ input image to compute a single $N \times M$ channel of the output image, to which the lth bias value is then

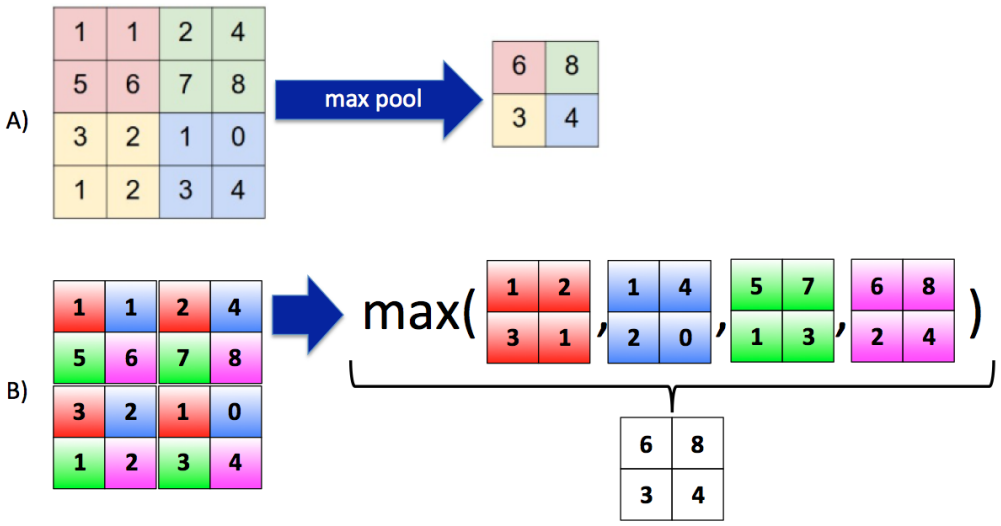

Figure 1: Example of using maxpool to reduce a 4x4 image to a 2x2 image. A) Each nonoverlapping 2x2 block of the image is replaced by the max over the 4 values in that block. B) In an alternative way of doing the computation, perhaps more suitable for Matlab implementation, the 4x4 image is converted into four 2x2 images by taking every other pixel value, starting at locations (1,1) , (1,2) , (2,1) and (2,2). A final 2x2 image is produced by taking the max over corresponding pixel locations in those 4 reduced images. Note: although this is conceptually what you want to do, in practice you can't apply max to 4 arrays at the same time, only 2 at a time.

added. Because there are D_2 filters (and bias values), this means the output image will have D_2 channels, and will therefore be of size $N \times M \times D_2$. See Figure 2.

Let F_l be the lth filter and b_l be the lth scalar bias value. We compute the lth channel of the output image using summations of 2D convolutions, defined as follows:

$$\text{Out}(:,:,l) \;=\; \sum_{k=1}^{D_1} F_l(:,:,k) * \text{In}(:,:,k) + b_l \quad .$$

In words, each channel $1 \leq k \leq D_1$ of the input image is convolved with the corresponding channel of the filter, those D_1 result images are summed up, and then the scalar bias value is added. This process is carried out D_2 times, for $1 \leq l \leq D_2$, to fill in all D_2 channels of the output image. When we say "convolution" here, we really mean convolution, for example using imfilter with the 'conv' option; the filters you will be given will not work correctly if you use cross-correlation. In addition, you want the size of each convolved image to remain $N \times M$, and you also want to use zero-padding.

Fully Connected: Input is an array of size $N \times M \times D_1$ and output is an array of size $1 \times 1 \times D_2$. Similar to the convolution layer, the fully connected layer applies a filter bank of D_2 linear filters and D_2 scalar bias values to compute output values. However, unlike convolution layers, each filter is the same size as the input image, and is applied in a way that computes a single, scalar valued output. Because there are D_2 filters and bias values, the output image will contain D_2 scalar values, which we will represent as a $1 \times 1 \times D_2$ array. See Figure 2. The name "fully connected" comes from the traditional neural network literature – each scalar output value is computed as a linear

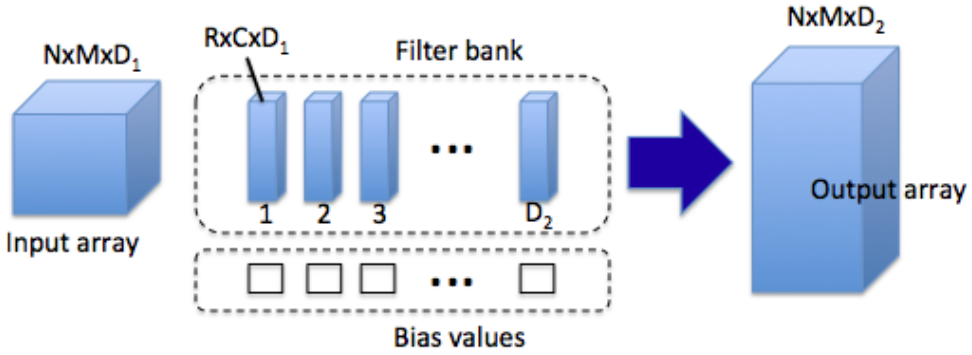

Figure 2: A convolution layer transforms an $N \times M \times D_1$ input array into an $N \times M \times D_2$ output array by convolving with a filter bank of D_2 linear filters and adding bias values. See text for details.

combination of the full set of $N \times M \times D_1$ input values.

Given an $N \times M \times D_1 \times D_2$ filter bank array and set of D_2 scalar bias values, let F_l be the lth filter and b_l be the lth bias value. We compute the lth channel of the output image as the scalar value:

$$\text{Out}(1,1,l) \;=\; \sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{k=1}^{D_1} F_l(i,j,k) \times \text{In}(i,j,k) + b_l \;\; .$$

In words, the lth output value is computed by multiplying every value in the input image by a corresponding filter weight, summing all of those terms up, and then adding the bias value. This is repeated for each filter F_l and bias b_l, $1 \leq l \leq D_2$, to compute all D_2 output values. Since each filter has the same number of values in it as the input image, if you were to rearrange each of them into a 1D vector of length NMD_1 you would recognize the equation above as performing a dot product operation followed by adding a scalar bias value. Whether or not you choose to compute it that way is up to you.

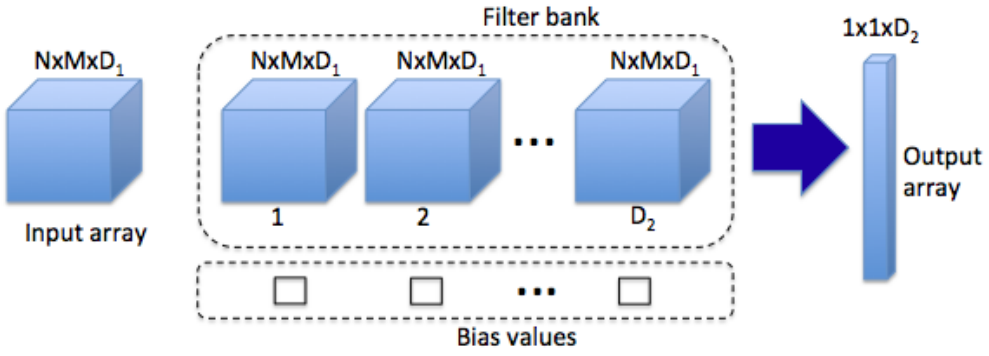

Figure 3: A fully connected layer transforms an $N \times M \times D_1$ input array into a $1 \times 1 \times D_2$ output array (basically a vector of numbers) by applying a filter bank of D_2 linear filters and additive bias values. Applying each filter is similar to a performing a dot product operation. See text for details.

Softmax: Input is an array of size $1 \times 1 \times D$ and output is an array of the same size. Each value in the output array is defined as

$$\text{Out}(1,1,k) = \frac{\exp(\text{In}(1,1,k) - \alpha)}{\sum_{k=1}^{D} \exp(\text{In}(1,1,k) - \alpha)}$$

where

$$\alpha = \max_k \text{In}(1,1,k)$$

is the maximum across all values in the input vector. Softmax takes a vector of arbitrary real numbers and converts them into numbers that can be viewed as probabilities, that is, all will lie between 0 and 1 and sum up to 1. Usually you will see softmax defined mathematically without the α term. However, you want to use our definition when writing a program, as it is numerically more stable.

3 Putting It All Together

Now that we know the basic building block functions, it is time to put them all together to do object recognition. A CNN takes a color image as input and applies a particular ordered sequence of the basic operations above, step by step, to ultimately produce a vector of class probabilities. Each operation is one "layer" of the computational chain, with each operation taking as input the output from the previous layer. The number of layers is called the "depth" of the CNN. Because the layer computations are all fairly simple, you need a lot of layers to accomplish anything interesting, and thus working CNNs tend to be deep (have a lot of computational depth). This is why training them is called deep learning.

We will implement an 18-layer CNN that takes a 32x32 color image as input and produces 10 object class probabilities as output. The images we will be using come from the well-known CIFAR-10 dataset. Despite being a "toy" dataset (thumbnail-sized images; only 10 object classes; only one object per image), results on CIFAR are often presented by deep learning practitioners because the dataset is so familiar to everyone in the field. The object classes and some sample images from each class are shown in Figure 4.

In the project directory, you will find a file called "cifar10testdata.mat" that contains a set of 10000 test images, 1000 from each class, along with their ground truth object classifications. Some sample code that illustrates reading in and dealing with the data format is shown below.

```
%loading this file defines imageset, trueclass, and classlabels
load 'cifar10testdata.mat'

%some sample code to read and display one image from each class
for classindex = 1:10
    %get indices of all images of that class
    inds = find(trueclass==classindex);
    %take first one
```

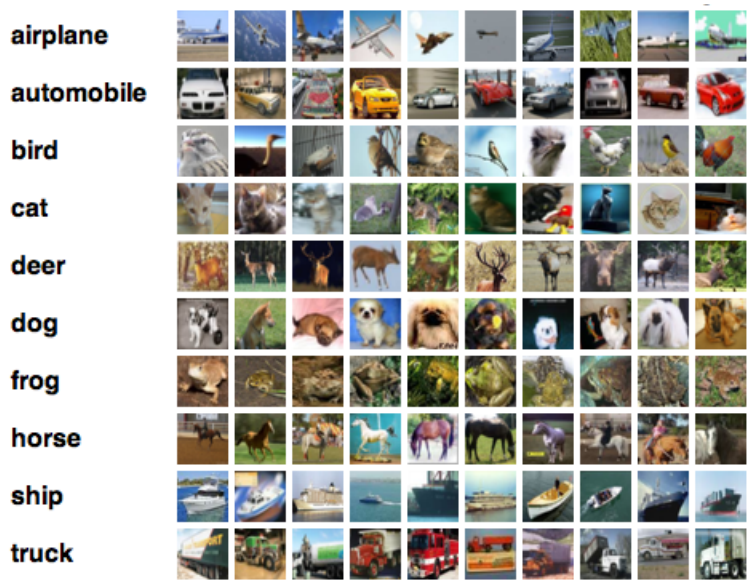

Figure 4: Sample images from each of the 10 object classes in CIFAR-10.

```
    imrgb = imageset(:,:,:,inds(1));
    %display it along with ground truth text label
    figure; imagesc(imrgb); truesize(gcf,[64 64]);
    title(sprintf('\%s',classlabels{classindex}));
end
```

Table 1 specifies each of the layer computations that our CNN will perform. Sizes of each input and output array are shown, and for the convolution and fully connected layers the size of the filter bank and number of scalar bias values are also shown. Note that the bulk of the computations consist of conv-ReLU pairs, with an occasional maxpool thrown in to progressively reduce the image sizes being processed. Normalize is done once at the beginning to prepare the input RGB image for processing; one fully connected layer is done at the end to distill all the results down to 10 numbers that softmax then converts into class probabilities.

The filter bank and bias values needed to completely specify the CNN are given to you in a file called "CNNparameters.mat" . Loading this file defines two variables, filterbanks and biasvectors. These are cell arrays (indexed with curly braces) because the stored arrays in filterbanks can have different sizes. For ease of use, there is an entry for each of the 18 layers of the CNN. However, only the cells for layers that involve using filterbank and bias values (the convolution layers and fully connected layer) have values stored in them, the rest are empty. The following sample code illustrates how to read and access the filterbanks and bias vectors.

```
%loading this file defines filterbanks and biasvectors
load 'CNNparameters.mat'

%sample code to verify which layers have filters and biases
for d = 1:length(filterbanks)
    filterbank = filterbanks{d};
```

layer #	layer type	input size	filterbank size	num biases	output size
1	normalize	$32 \times 32 \times 3$			$32 \times 32 \times 3$
2	convolve	$32 \times 32 \times 3$	$3 \times 3 \times 3 \times 10$	10	$32 \times 32 \times 10$
3	ReLU	$32 \times 32 \times 10$			$32 \times 32 \times 10$
4	convolve	$32 \times 32 \times 10$	$3 \times 3 \times 10 \times 10$	10	$32 \times 32 \times 10$
5	ReLU	$32 \times 32 \times 10$			$32 \times 32 \times 10$
6	maxpool	$32 \times 32 \times 10$			$16 \times 16 \times 10$
7	convolve	$16 \times 16 \times 10$	$3 \times 3 \times 10 \times 10$	10	$16 \times 16 \times 10$
8	ReLU	$16 \times 16 \times 10$			$16 \times 16 \times 10$
9	convolve	$16 \times 16 \times 10$	$3 \times 3 \times 10 \times 10$	10	$16 \times 16 \times 10$
10	ReLU	$16 \times 16 \times 10$			$16 \times 16 \times 10$
11	maxpool	$16 \times 16 \times 10$			$8 \times 8 \times 10$
12	convolve	$8 \times 8 \times 10$	$3 \times 3 \times 10 \times 10$	10	$8 \times 8 \times 10$
13	ReLU	$8 \times 8 \times 10$			$8 \times 8 \times 10$
14	convolve	$8 \times 8 \times 10$	$3 \times 3 \times 10 \times 10$	10	$8 \times 8 \times 10$
15	ReLU	$8 \times 8 \times 10$			$8 \times 8 \times 10$
16	maxpool	$8 \times 8 \times 10$			$4 \times 4 \times 10$
17	fullconnect	$4 \times 4 \times 10$	$4 \times 4 \times 10 \times 10$	10	$1 \times 1 \times 10$
18	softmax	$1 \times 1 \times 10$			$1 \times 1 \times 10$

Table 1: An 18-layer convolutional neural net.

```
    if not(isempty(filterbank))
        fprintf('layer %d has filters and biases\n',d);
        fprintf('    filterbank size %d x %d x %d x %d\n', ...
            size(filterbank,1),size(filterbank,2), ...
            size(filterbank,3),size(filterbank,4));
        biasvec = biasvectors{d};
        fprintf('    number of biases is %d\n',length(biasvec));
    end
end
```

Finally, to help you develop your code correctly, the file "debuggingTest.mat" contains a sample test image (imrgb) and expected output array produced by each layer of the CNN (layerResults) when that image is used as input. The following sample code illustrates how to read and access this data. I highly recommend verifying that your implementation of each of the algorithmic building blocks described above works as it is supposed to by comparing its output to these known correct results. For example, if you have just finished implementing a maxpool routine and want to test it, make sure that when it is run on the image array in layerResults{5}, it produces the output array in layerResults{6}. When all types of layers have been implemented, you should be able to start with imrgb and run the whole CNN from end to end to get the probabilities in layerResults{18}.

```
%loading this file defines imrgb and layerResults
load 'debuggingTest.mat'
```

```
%sample code to show image and access expected results
figure; imagesc(imrgb); truesize(gcf,[64 64]);
for d = 1:length(layerResults)
    result = layerResults{d};
    fprintf('layer %d output is size %d x %d x %d\n',...
        d,size(result,1),size(result,2), size(result,3));
end
%find most probable class
classprobvec = squeeze(layerResults{end});
[maxprob,maxclass] = max(classprobvec);
%note, classlabels is defined in  'cifar10testdata.mat'
fprintf('estimated class is %s with probability %.4f\n',...
    classlabels{maxclass},maxprob);
```

4 Quantitative Evaluation

How do you know that your algorithm works correctly? There are two meanings of *correct* we could be interested in. One is whether your code is implemented correctly, which you can check by using the information in 'debuggingTest.mat' to verify that each layer of your CNN is producing the expected results. However, in real life (industry coding; grad school research) it is taken as a given that you are competent enough to implement a program that does what the specifications say it should do. The question then is how well does the method you are implementing perform? This is where the topic of performance evaluation comes in.

There are many ways to evaluate a classifier. What we perhaps most care about in a classifier is overall accuracy, that is, what percentage of test examples does it choose the correct class for. Other questions we might be interested in are whether the approach finds some classes to be easier to classify than others, and whether there are pairs of classes that are frequently confused for each other. All of these questions can be answered by compiling a *confusion matrix* or *contingency table* on a representative test set.

The obvious way for having our CNN classifier choose what object class is in the image is to take the class associated with the highest probability in the output class probability vector (using some heuristic to break ties if necessary). Using the 10000 test image examples in 'cifar10testdata.mat', we'd like you to classify each one and accumulate a 10×10 table A_{ij} where each table entry contains the number of times that an example with a ground truth class of i was classified as object class j by the CNN. Values along the diagonal of this table represent examples that were classified correctly (e.g. the image had true class 'airplane' and was classified as 'airplane'). All examples off the diagonal are errors. The *classification accuracy* or *classification rate* of the CNN can then be computed as

$$\text{Accuracy} = \frac{\sum_i A_{ii}}{\sum_i \sum_j A_{ij}} .$$

Further examination of the confusion matrix can highlight classes that seems easy to classify cor-

rectly (rows where the diagonal is a large percentage of the row sum), and pairs of classes that are frequently confused for each other (large off-diagonal values).

Another performance evaluation measure that might be interesting to try is to plot a curve showing classification accuracy with respect to the top-k classes. This would be a plot where the x axis ranges from 1 to 10, and the y axis ranges from 0 to 100 percent. For each integer value k on the x-axis, you would compute and plot as $y(k)$ the percentage of times that the correct object class appeared in the top-k ranked classes, as determined by probability scores sorted from highest to lowest. The plotted value for $k = 1$ is just the usual classification accuracy described above. The plotted value for $k = 2$ is the percentage of times that the correct class for an image is one of the 2 classes that have the highest computed probability scores for that image. And so on. For $k = 10$ the curve would be at 100 percent accuracy, obviously.

5 Grading

Half of your grade will be based on submitting a runable program, and the other half will be based on a written report discussing your program, design decisions, and experimental observations. In particular, we want you to turn in the following two things:

1. Submit a written report in which you discuss at least the following:

a) Summarize *in your own words* what you think the project was about. What were the tasks you performed; what did you expect to achieve?

b) Present an outline of the procedural approach along with a flowchart showing the flow of control and subroutine structure of your Matlab code. Explain any design decisions you had to make. Even though the mathematical specification of each part of this project is fairly strict, there are a lot of different ways you could implement each building block in Matlab, ranging from C-like nested for-loop computations, to cleverly vectorized code. Be sure to document any deviations you made from the above project descriptions (and why), or any additional functionality you added to increase robustness or generality of the approach.

c) Experimental observations. What do you observe about the behavior of your program when you run it? Does it seem to work the way you think it should? Run the different portions of your code and show pictures of intermediate and final results that convince us that the program does what you think it does. Each channel of an intermediate result array in the CNN can be interpreted as a greyscale image. So, for example, the output from layer 2 of the CNN is an array of size $32 \times 32 \times 10$, and this could be displayed as 10 small greyscale images, each of size 32×32. CNN practitioners often display these intermediate results and interpret them as images showing the results of learned "feature" detectors. The results from the final softmax layer could be displayed as a bar chart.

d) Run performance evaluation experiments as described above to compute the confusion matrix, classification rate and top-k classification plot. Discuss.

e) If you are in an exploratory mood, find some images of your own on the web and input them to the CNN to see how well it does on them. To do this, you will need to reduce the image size down to a $32 \times 32 \times 3$ color thumbnail image. How would you do this in Matlab? (hint: we discussed generating thumbnail images in one of the lectures). Of course, you will get best results if the images you choose actually contain one of the object classes, and that object should pretty much dominate the image. What happens if you give it an image containing an object that it doesn't know about? For example, what happens if you input an image of your face? (no matter what the output classification for your face is, I'm sure it will be hilarious). Can you think of a test that you could perform on the output probabilities to extend the classification to include an "unknown" category.

f) Document what each team member did to contribute to the project. It is OK if you divide up the labor into different tasks (actually it is expected), and it is OK if not everyone contributes precisely equal amounts of time/effort on each project. However, if two people did all the work because the third team member could not be contacted until the day before the project was due, this is where you get to tell us about it, so the grading can reflect the inequity.

FAQ: How long should your report be? We don't have a strict number of pages in mind, but as a general rule of thumb, if the length of your report (excluding figures) is less than the length of this project description document, you probably haven't included enough detail about what you did/observed.

2. Turn in a running version of your main program and all subroutines in a single zip or tar archive file (e.g. put all code, subroutines, etc in a single directory, then make a zip file of that directory for submission via Angel). We can then unzip/untar everything and go from there. Include one or more demo routines that can be invoked with no arguments and that load any input needed from the local directory and display intermediate and final results that show the different portions of your program are working as intended. We won't necessarily be running the code ourselves on other input, but put enough comments in the code so that we know how to run it if we want to spot check anything we think looks odd. The more thought and effort you put in to demonstrating/illustrating in your written report that your code works correctly, the less likely we will be to feel the need to poke around in your code.

6 References

`http://cs231n.stanford.edu/syllabus.html`
Stanford course on Convolutional Neural Networks for Visual Recognition, taught by Andrej Karpathy and Fei-Fei Li. This project was largely inspired by the contents of that course.

`http://www.cs.toronto.edu/~kriz/cifar.html`
CIFAR-10 and CIFAR-100 datasets. Our test dataset is the 10000 image "test batch" of CIFAR-10.