CSE 152: Introduction to Computer Vision
Homework 4: Deep Learning

Instructions:

- Total points: 100
- **Due: 11:59 pm, Friday, Dec 7, 2018**
- For the implementation part, we've provided you with a number of functions and scripts in the hopes of alleviating some tedious or error-prone sections of the implementation. **In general, you should try to implement your part under the comment line "YOUR CODE HERE," and stick to the headers and file conventions provided. However, don't let this limit you – you are free to modify any code as needed. If you make any nontrivial code changes outside of the "YOUR CODE HERE" or otherwise sanctioned regions, document it in your writeup.**
- When you're finished, compress your code and writeup into a single file named **LastName_FirstName_YourPID_hw4.zip** and submit it to descartes.ucsd.edu. **Also submit your writeup (pdf) to Gradescope.**
- If you do not submit to both Gradescope and the server, we reserve the right to apply a 30% penalty.
- A complete submission consists of the following files (do not hand in any networks, training data, or MatConvNet itself!)
  - Everything in the `code/` directory, but most importantly `part1.m`, `part1_cnn_init.m`, `part1_setup_data.m`, and any other code you modify
  - A writeup (PDF format)

  Make sure that we can run your code without any modification (assuming we have the training data and MatConvNet), otherwise you are at risk of losing points.
- **Start early!** Please familiarize yourself with MATLAB and allow enough time for debugging.

This project is an introduction to deep learning tools for computer vision. we will design and train deep convolutional networks for scene recognition using the MatConvNet toolbox.

In the project, we will train a deep convolutional network to recognize scenes. The starter code provides a simple network architecture (which does not work that well), and we will add jittering, normalization, regularization, and more layers to increase recognition accuracy to 50, 60, or perhaps 70%.

1 Starter Code Outline

The following is an outline of the stencil code:

- part1.m. The top level function for training a deep network from scratch for scene recognition. If you run this starter code unmodified, then it will train a simple network that achieves only 25% accuracy.

  part1.m calls:

  - part1_setup_data.m . Loads the 15 scene database into MatConvNet imdb format.
  - part1_cnn_init.m. Initializes the convolutional neural network by specifying the various layers which perform convolution, max pooling, non-linearities, normalization, regularization, and the final loss layer.
  - part1_get_batch() (defined inside part1.m) This operates on each batch of training images to be passed into the network during training. This is where you can "jitter" your training data.

The network training will be performed by cnn_train.m, which in turn calls vl_simplenn.m. You will not need to modify those functions for this homework.

2 Part 0: Setup

There are two approaches to installing MatConvNet. One is to manually install everything yourself. This is probably the more tedious option, so let's avoid that if possible. The first three lines of part0.m should perform all of the MatConvNet installation automatically (if all goes well). So as a first attempt, try to run the part0.m script and hope that everything works.

part0.m will perform installation and run a simple MatConvNet test case, which involves downloading a pre-trained CNN for classification and running it on a test image. Before you proceed to step 1, you should verify your MatConvNet installation by making sure you can run part0.m correctly.

(It might take a while to download the 233MB VGG-F network used in the demo.)

2.1 Manual installation

Hopefully part0.m will succeed in installing MatConvNet on its own. But if installation fails, you may need to download MatConvNet yourself. You must separately download MatConvNet 1.0 beta 16. (Click **here** for a direct link to beta 16.) Note that the homework depends on having this version of MatConvNet, so you should install this one; any other versions are not officially supported.

MatConvNet isn't precompiled like VLFeat, so we compiled it for you. Here are the mex files for CPU-only usage for 64bit Windows, MacOS, and Linux: **MatConvNet64mex.zip (.5 MB)**. The files go in [MatConvNet-Path]/matlab/mex/.

Once you have performed the installation, comment out the three installation lines at the top of part0.m and again confirm that you can run the pre-trained CNN correctly (by running part0.m to completion).

2.2 Troubleshooting

If you encounter errors trying to run part0.m, make sure:

1. You have MatConvNet 1.0 beta 16 (not a later version).
2. You download imagenet-vgg-f.mat from the link in part0.m and not from MatConvNet (because it was changed for beta 17 and it is not backwards compatible).
3. If you encounter the error "Error using mex; No supported compiler was found," visit `https://www.mathworks.com/support/compilers.html` to to make sure that the correct compilers are installed. Also if necessary, run "mex -setup C++" to ensure that MEX is configured for C++ compilation not for C.
4. Your mex files are in the correct location [MatConvNetPath]/matlab/mex/. If you encounter errors about invalid mex files in Windows you may be missing Visual C++ Redistributable Packages. If you encounter an error about about labindex being undefined you may be missing the Parallel Computing Toolbox for Matlab.
5. If you are missing the Parallel Computing Toolbox, you can go to the Home tab on the bar at the top of MATLAB and click Add-Ons > Get Add-Ons. Do a search for "parallel," click on the Parallel Computing Toolbox, and choose the option to (sign in and) install.

Before we start building our own deep convolutional networks, it might be useful to have a look at MatConvNet's tutorial (**here**). In particular, you should be able to understand Part 1 of the tutorial. In addition to the examples shown in parts 3 and 4 of the tutorial, MatConvNet has example code (**here**) for training networks to recognize the MNIST and CIFAR datasets. Your project

follows the same outline as those examples. Feel free to take a look at that code for inspiration. You can run the example code to watch the training process for MNIST and CIFAR. Training will take about 5 and 15 minutes for those datasets, respectively.

Compiling MatConvNet with GPU support is more complex and not needed for this project.

3 Part 1: Training a deep neural network

In this part, we will run part1.m to do end-to-end learning by a neural network, in which a highly non-linear representation is learned for our data to maximize our objective (in this case, 15-way classification accuracy).

First, let's take a look at the network architecture used in this experiment. Here is the code from part1_cnn_init.m that specifies the network structure:

```
net.layers = {} ;
net.layers{end+1} = struct('type', 'conv', ...
                           'weights', {{f*randn(9,9,1,10, 'single'), zeros(1, 10, 'single')}}, ...
                           'stride', 1, ...
                           'pad', 0, ...
                           'name', 'conv1') ;
net.layers{end+1} = struct('type', 'pool', ...
                           'method', 'max', ...
                           'pool', [7 7], ...
                           'stride', 7, ...
                           'pad', 0) ;
net.layers{end+1} = struct('type', 'relu') ;
net.layers{end+1} = struct('type', 'conv', ...
                           'weights', {{f*randn(8,8,10,15, 'single'), zeros(1,15,'single')}}, ...
                           'stride', 1, ...
                           'pad', 0, ...
                           'name', 'fc1') ;
net.layers{end+1} = struct('type', 'softmaxloss') ;
```

Let's make sure we understand what's going on here. This simple baseline network has 4 layers: a convolutional layer, followed by a max pool layer, followed by a rectified linear layer, followed by another convolutional layer. This last convolutional layer might be called a "fully connected" or "fc" layer because its output has a spatial resolution of 1x1. Equivalently, every unit in the output of that layer is a function of the entire previous layer (thus "fully connected"). But mathematically, there's not really any difference from "convolutional" layers so we specify them in the same way in MatConvNet.

Let's look at the first convolutional layer. The "weights" are the filters being learned. They are initialized with random numbers from a Gaussian distribution. The inputs to randn(9,9,1,10) mean the filters have a 9x9 spatial resolution, span 1 filter depth (because the input images are grayscale), and that there are 10 filters. The network also learns a bias or constant offset to associate with the output of each filter. This is what zeros(1,10) initializes.

The next layer is a max pooling layer. It will take a max over a 7x7 sliding window and then subsample the resulting image / map with a stride of 7. Thus the max pooling layer will decrease the spatial resolution by a factor of 7

according to the stride parameter. The filter depth will remain the same (10). There are other pooling possibilities (e.g. average pooling) but we will only use max pooling in this project.

The next layer is the non-linearity. Any values in the feature map from the max pooling layer which are negative will be set to 0. There are other non-linearity possibilities (e.g. sigmoid) but we will use only ReLU in this project.

Note that the pool layer and ReLU layer have no learned parameters associated with them.

Finally, we have the last layer which is convolutional (but might be called "fully connected" because it happens to reduce the spatial resolution to 1x1). The filters learned at this layer operate on the rectified, subsampled, maxpooled filter responses from the first layer. The output of this layer must be 1x1 spatial resolution (or "data size") and it must have a filter depth of 15 (corresponding to the 15 categories of the 15 scene database). This is achieved by initializing the weights with randn(8,8,10,15). 8x8 is the spatial resolution of the filters. 10 is the number of filter dimensions that each of these filters take as input and 15 is the number of dimensions out. 10 is highlighted in green to emphasize that it must be the same in those 3 places – if the first convolutional layer has weights for 10 filters, it must also have offsets for 10 filters, and the next convolutional layer must take as input 10 filter dimensions.

At the top of our network we add one more layer which is only used for training. This is the "loss" layer. There are many possible loss functions but we will use the "softmax" loss for this project. This loss function will measure how badly the network is doing for any input (i.e. how different its final layer activations are from the ground truth, where ground truth in our case is category membership). The network weights will update, through backpropagation, based on the derivative of the loss function. With each training batch the network weights will take a tiny gradient descent step in the direction that should decrease the loss function (but isn't actually guaranteed to, because the steps are of some finite length, or because dropout will turn off part of the network).

How did we know to make the final layer filters have a spatial resolution of 8x8? It's not obvious because we don't directly specify output resolution. Instead it is derived from the input image resolution and the filter widths, padding, and strides of the previous layers. Luckily MatConvNet provides a visualization function vl_simplenn_display to help us figure this out. Here is what it looks like if we specify the net as shown above and then call vl_simplenn_display(net, 'inputSize', [64 64 1 50]).

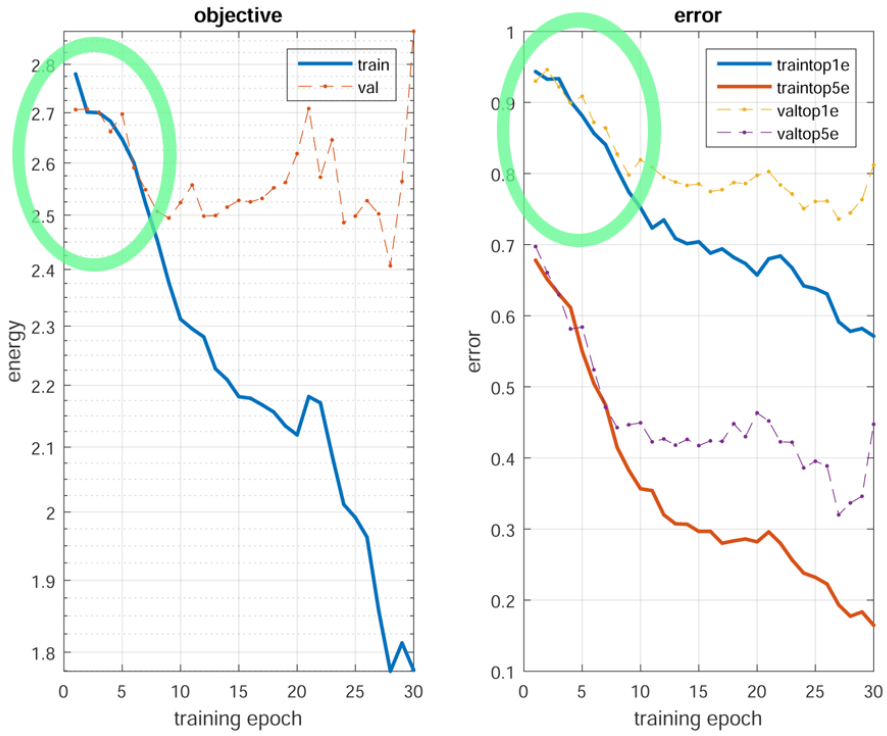
```
     layer|    0|    1|    2|    3|   4|      5|
      type|input| conv|mpool| relu|conv|softmxl|
      name|  n/a|conv1|     |     | fc1|       |
----------|-----|-----|-----|-----|----|-------|
   support|  n/a|    9|    7|    1|   8|      1|
  filt dim|  n/a|    1|  n/a|  n/a|  10|    n/a|
 num filts|  n/a|   10|  n/a|  n/a|  15|    n/a|
    stride|  n/a|    1|    7|    1|   1|      1|
       pad|  n/a|    0|    0|    0|   0|      0|
----------|-----|-----|-----|-----|----|-------|
   rf size|  n/a|    9|   15|   15|  64|     64|
 rf offset|  n/a|    5|    8|    8|32.5|   32.5|
 rf stride|  n/a|    1|    7|    7|   7|      7|
----------|-----|-----|-----|-----|----|-------|
 data size|   64|   56|    8|    8|   1|      1|
data depth|    1|   10|   10|   10|  15|      1|
  data num|   50|   50|   50|   50|  50|      1|
----------|-----|-----|-----|-----|----|-------|
  data mem|800KB|  6MB|125KB|125KB| 3KB|     4B|
 param mem|  n/a|  3KB|   0B|   0B|38KB|     0B|

parameter memory|41KB (1e+04 parameters)|
     data memory|7MB (for batch size 50)|
```

If the last convolutional layer had a filter size of 6x6 that would lead to a "data size" in the network visualization of 3x3 and we would know we need to change things (subsample more in previous layers or create wider filters in the final layer). In general it is not at all obvious what the right network architecture is. It takes a lot of tricks to design the right network and training strategy for optimal performance.

We just said the network has 4 real layers but this visualization shows 6. That's because it includes a layer 0 which is the input image and a layer 5 which is the loss layer. For each layer this visualization shows several useful attributes. "data size" is the spatial resolution of the feature maps at each level. In this network and most deep networks this will decrease as you move up the network. "data depth" is the number of channels or filters in each layer. This will tend to increase as you move up a network. "rf size" is the receptive field size. That is how large an area in the original image a particular network unit is sensitive to. This will increase as you move up the network. Finally this visualization shows us that the network has 10,000 free parameters, the vast majority of them associated with the last convolutional layer.

OK, now we understand a bit about the network. Let's analyze its performance. After 30 training epochs (30 passes through the training data) Matlab's Figure 1 should look like this:



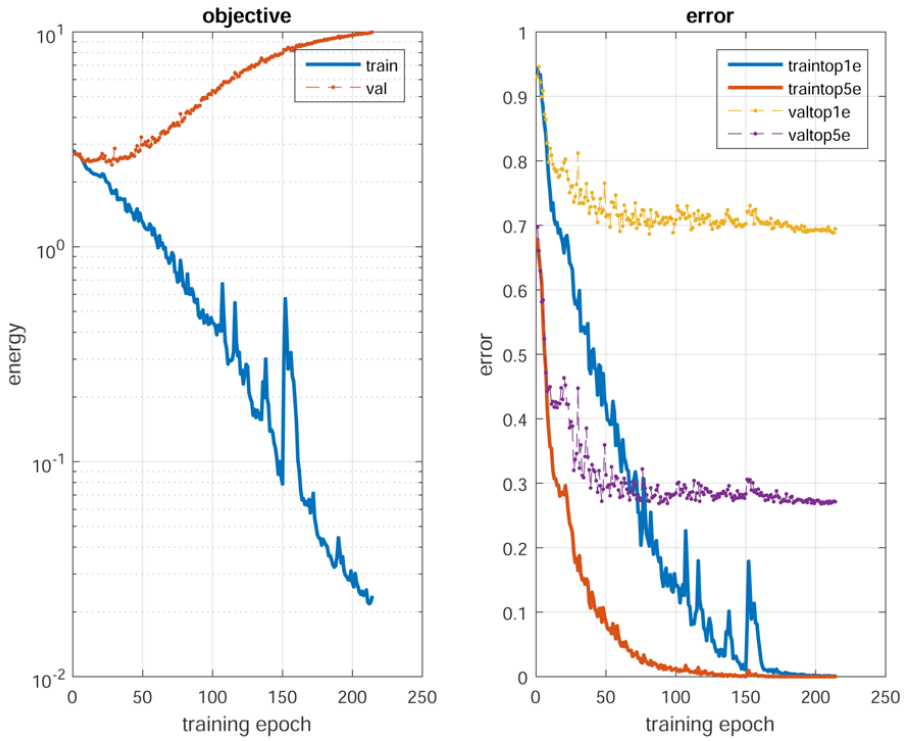
The left pane shows the training error (blue) and validation error (dashed orange) across training epochs. Each training epoch is a pass over the entire training set of 1500 images broken up into "batches" of 50 training instances. The code shuffles the order of the training instances randomly each epoch. When the network makes mistakes, it incurs a "loss" and backpropagation updates the weights of the network in a direction that should decrease the loss. Therefore the blue line should more or less decrease monotonically. On the other hand, the orange dashed line is the error incurred on the held out test set. The figure refers to it as "val" or "validation". In a realistic recognition scenario we might have three sets of data: train, validation, and test. We would use validation to assess how well our training is working and to know when to stop training and then we would test on a completely held out test set. For this project the validation set is our test set. We're trying to maximize performance on the validation set and that's it. The pass through the validation set does not change the network weights in any way. The pass through the validation set is also 3 times faster than the training pass because it does not have the "backwards" pass to update network weights.

The right pane shows the training and testing accuracy on the train and test (val) data sets across the same training epochs. It shows top 1 error – how often the highest scoring guess is wrong – and top 5 error – how often all of the 5 highest scoring guesses are wrong. We're interested in top 1 error, specifically

the top 1 error on the held out validation / test set.

In this experiment, the training and test top 1 error started out around 93% which is exactly what we would expect. If you have 15 categories and you make a random guess on each test case, you will be wrong 93% of the time. As the training progressed and the network weights moved away from their random initialization, accuracy increased.

Note that the areas circled in green corresponding to the first 8 training epochs. During these epochs, the training and validation error were decreasing which is exactly what we want to see. Beyond that point the error on the training dataset kept decreasing, but the validation error did not. Our lowest error on the validation/test set is around 75% (or 25% accuracy). We are overfitting to our training data. This is hard to avoid with a small training set. In fact, if we let this experiment run for 200 epochs we see that it is possible for the training accuracy to become perfect with no appreciable increase in test accuracy:



Now, we are going to take several steps to improve the performance of our convolutional network. The modifications we make in Part 1 will familiarize you with the building blocks of deep learning that can lead to impressive performance with enough training data. In the end, you might decide that this isn't any simpler than hand-designing a feature. Also, with the relatively small amount of training data in the 15 scene database, it is very hard to outperform hand-

crafted features.

Learning rate. Before we start making changes, there is a very important learning parameter that you might need to tune any time you change the network or the data being input to the network. The learning rate (set by default as opts.LearningRate = 0.0001 in part1.m) determines the size of the gradient descent steps taken by the network weights. If things aren't working, try making it much smaller or larger (e.g. by factors of 10). If the objective remains exactly constant over the first dozen epochs, the learning rate might have been too high and "broken" some aspect of the network. If the objective spikes or even becomes NaN then the learning rate may also be too large. However, a very small learning rate requires many training epochs.

3.1 Problem 1: We don't have enough training data. Let's "jitter".

If you left-right flip (mirror) an image of a scene, it never changes categories. A kitchen doesn't become a forest when mirrored. This isn't true in all domains – a "d" becomes a "b" when mirrored, so you can't "jitter" digit recognition training data in the same way. But we can synthetically increase our amount of training data by left-right mirroring training images during the learning process.

The learning process calls getBatch() in part1.m each time it wants training or testing images. Modify getBatch() to randomly flip some of the images (or entire batches). Useful functions: rand and fliplr.

You can try more elaborate forms of jittering – zooming in a random amount, rotating a random amount, taking a random crop, etc. Mirroring helps quite a bit on its own, though, and is easy to implement. You should see a roughly 10% increase in accuracy by adding mirroring.

After you implement mirroring, you should notice that your training error doesn't drop as quickly. That's actually a good thing, because it means the network isn't overfitting to the 1,500 original training images as much (because it sees 3,000 training images now, although they're not as good as 3,000 truly independent samples). Because the training and test errors fall more slowly, you may need more training epochs or you may try modifying the learning rate.

3.2 Problem 2: The images aren't zero-centered.

One simple trick which can help a lot is to subtract the mean from every image. Modify part1_setup_data.m so that it computes the mean image and then subtracts the mean from all images before returning imdb. It would arguably be more proper to only compute the mean from the training images (since the test/validation images should be strictly held out) but it won't make much of a difference. After doing this you should see another 15% or so increase in accuracy.

3.3 Problem 3: Our network isn't regularized.

If you train your network (especially for more than the default 30 epochs) you'll see that the training error can decrease to zero while the val top1 error hovers at 40% to 50%. The network has learned weights which can perfectly recognize the training data, but those weights don't generalize to held out test data. The best regularization would be more training data but we don't have that. Instead we will use dropout regularization. We add a dropout layer to our convolutional net as follows:

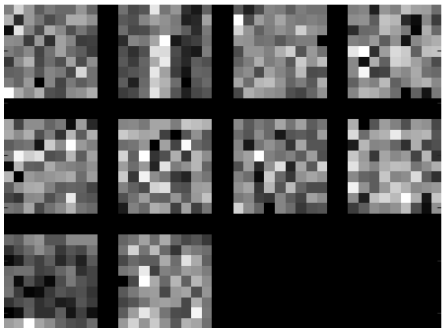
```
net.layers{end+1} = struct('type', 'dropout', ...
                           'rate', 0.5) ;
```

What does dropout regularization do? It randomly turns off network connections at training time to fight overfitting. This prevents a unit in one layer from relying too strongly on a single unit in the previous layer. Dropout regularization can be interpreted as simultaneously training many "thinned" versions of your network. At test test, all connections are restored which is analogous to taking an average prediction over all of the "thinned" networks. You can see a more complete discussion of dropout regularization in this paper.

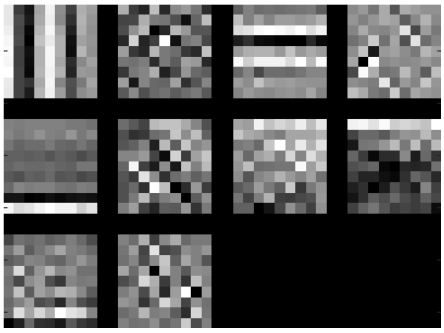
The dropout layer has only one free parameter – the dropout rate – the proportion of connections that are randomly deleted. The default of 0.5 should be fine. Insert a dropout layer between your convolutional layers. In particular, insert it directly before your last convolutional layer. Your test accuracy should increase by another 10%. Your train accuracy should decrease much more slowly. That's to be expected: you're making life much harder for the training algorithm by cutting out connections randomly.

If you increase the number of training epochs (and maybe decrease the learning rate) you should be able to achieve 60% test accuracy (40% top1 val) or slightly better at this point. Notice how much more structured the learned filters are at this point compared to the initial network before we made improvements:

First layer filters learned by starter code



First layer filters learned with mirroring, center, and dropout regularization



3.4 Problem 4: Our network isn't deep.

Let's take a moment to reflect on what our convolutional network is actually doing. We learn filters which seem to be looking for horizontal edges, vertical edges, and parallel edges. Some of the filters have diagonal orientations and some seem to be looking for high frequencies or center-surround. This learned filter bank is applied to each input image, the maximum response from each 7x7 block is taken by the max pooling, and then the rectified linear layer zeros out negative values. The fully connected layer sees a 10 channel image with 8x8 spatial resolution. It learns 15 linear classifiers (a linear filter with a learned threshold is basically a linear classifier) on this 8x8 filter response map.

Our convolutional network to this point isn't "deep". It has two layers with learned weights. Contrast this with the example networks for MNIST and CIFAR in MatConvNet which contain 4 and 5 layers, respectively. AlexNet and VGG-F contain 8 layers. The VGG "very deep" networks contain 16 and 19 layers. ResNet contains up to 150 layers.

One quite unsatisfying aspect of our current network architecture is that the max-pooling operation covers a window of 7x7 and then is subsampled with a stride of 7. That seems overly lossy and deep networks usually do not subsample by more than a factor of 2 or 3 each layer.

Let's make our network deeper by adding an additional convolutional layer in part1_cnn_init.m. In fact, we probably don't want to add just a convolutional layer, but another max-pool layer and ReLU layer, as well. For example, you might insert a convolutional layer after the existing ReLU layer with a 5x5 spatial support followed by a max-pool over a 3x3 window with a stride of 2. You can reduce the max-pool window in the previous layer, adjust padding, and reduce the spatial resolution of the final layer until vl_simplenn_display(net, 'inputSize', [64 64 1 50]), which is called at the end of part1_cnn_init() shows that your network's final layer (not counting the softmax) has a data size of 1 and a data depth of 15. You also need to make sure that the data depth output by any channel matches the data depth input to the following channel. For instance, maybe your new convolutional layer takes in the 10 channels of the first layer but outputs 15 channels. The final layer would then need to have its weights initialized accordingly to account for the fact that it operates on a 15 channel image instead of a 10 channel image.

We leave it up to you to decide the specifics of your slightly deeper network: filter depth, padding, max-pooling, stride, etc. The network will probably take longer to train because it will have more parameters and deeper networks take longer to converge. You might need to use more training epochs, but even then it will be difficult to outperform your shallow network. It is not required that your deeper network increases accuracy over the shallow network. As long as you can achieve **less than** 45% top1 validation error for some epoch (*over the entire validation set, not just for one batch*) with a deeper network which uses mirroring to jitter, zero-centers the images as they are loaded, and regularizes the network with a dropout layer you will receive full credit for this part.

See the next section for a full description of the code and performance re-

quirements for this homework.

4 Requirements (100 points total)

4.1 Code (5 pts for implementing each piece)

For full credit, you must implement some form of jittering (as per problem 1), some form of normalization (as per problem 2), some form of dropout regularization (as per problem 3), and a deeper network (as per problem 4).

4.2 Network Performance (45 pts for meeting overall benchmark, 20 pts for meeting deeper network benchmark)

Note: when we talk about validation error below, we mean the error across the entire validation set, not the error for one batch (as printed in the big log during training). In other words, we care about the valtop1e orange plot in the error graph, and the "Lowest validation error is" value printed at the end of training. Feel free to modify `cnn_train.m` to print info.val.error periodically throughout training, although this is in no way required.

Report your best top1 validation accuracy and include the error graph for it. *If your best top1 validation accuracy does not use a deeper network*, also report your best top1 validation accuracy for the deeper network and include the error graph for the deeper network variant.

For full credit, you must achieve sub-40% top1 validation error in some epoch for any network/training configuration, and sub-45% top1 validation error in some epoch for a *deeper network*.

Your network must be trained from scratch by you, using the data we provide. If you make any modifications or add any extensions past the minimum requirements of problems 1 through 4, document them in your writeup. However, it is possible to meet these performance requirements just by implementing problems 1 through 4 well.

4.3 Writeup Questions (5 pts apiece)

Also answer the following questions in your writeup.

1. Explain why subtracting the mean from our images improves the performance of our neural network.

2. Describe the role that pooling plays in the convolutional neural network. How does pooling change the number of parameters in the network, and what effects does this have?

3. What issues might arise in using sigmoids as activation functions? How does the rectified linear unit function address these issues? Can you describe problems that may arise with using ReLU?

5 Extra Credit

For every extra 5% of error you knock off past 40% top1 validation error, you will receive four extra credit points. For example, if you achieve sub-35% error, e.g. 34.8% error, you will receive four extra credit points. If you somehow achieve sub-20% error, you will receive 16 extra credit points. Again, you must train from scratch and stick to the data we provide. You must also document the extensions you make in your writeup (and show the effects they had on the results, e.g. by showing the performance graphs with and without the extension), and make sure that we can reproduce your results if we run your code.

Note that this is very hard, and trying to improve accuracy on this assignment is probably not the most effective way to raise your grade in this class (studying for the final is likely a better use of your time).

If you decide to pursue this, here are some potential avenues of improvement:

- If you look at MatConvNet's ImageNet examples you can see that the learning rate isn't constant during training. You can specify learning rate as pts.learningRate = logspace(-3, -5, 120) to have it change from .001 to .00001 over 120 training epochs, for instance. This can improve performance slightly.

- You can try increasing the filter depth of the network. The example networks for MNIST, CIFAR, and ImageNet have 20, 32, and 64 filters in the first layer and it tends to increase as you go up the network. Unfortunately, it doesn't seem to help too much in our case probably because of lack of training data.

- The MNIST, CIFAR, and ImageNet examples in MatConvNet show numerous advanced strategies: Use of normalization layers, variable learning rate per layer (the two elements of the per-layer learning rate in cnn_cifar_init.m are the relative learning rates for the filters and the bias terms), use of average pooling instead of max pooling for some layers, skipping ReLU layers between some convolutional layers, initializing weights with distributions other than randn, more dramatic jittering, etc.

- The more free parameters your network has the more prone to overfitting it is. Multiple dropout layers can help fight back against this, but will slow down training considerably.

- One obvious limitation of our network is that it operates on 64x64 images when the scene images are generally closer to 256x256. We're definitely losing valuable texture information by working at low resolution. Luckily, it is not necessarily slow to work with the higher resolution images if you put a greater-than-one stride in your first convolutional layer. The VGG-F network adopts this strategy. You can see in cnn_imagenet_init.m that its first layer uses 11x11 filters with a stride of 4. This is 1/16th as many evaluations as a stride of 1.

- The images can be normalized more strongly (e.g., making them have unit standard deviation) but this did not help in my experiments.
- You can try alternate loss layers at the top of your network. E.g. `net.layersend+1 = struct('name', 'hinge loss', 'type', 'loss', 'loss', 'mhinge')` for hinge loss.
- You can train the VGG-F network from scratch on the 15 scene database. You can call cnn_imagenet_init.m to get a randomly initialized VGG-F and train it just like your other networks. It works better than I would expect considering how little training data we have.