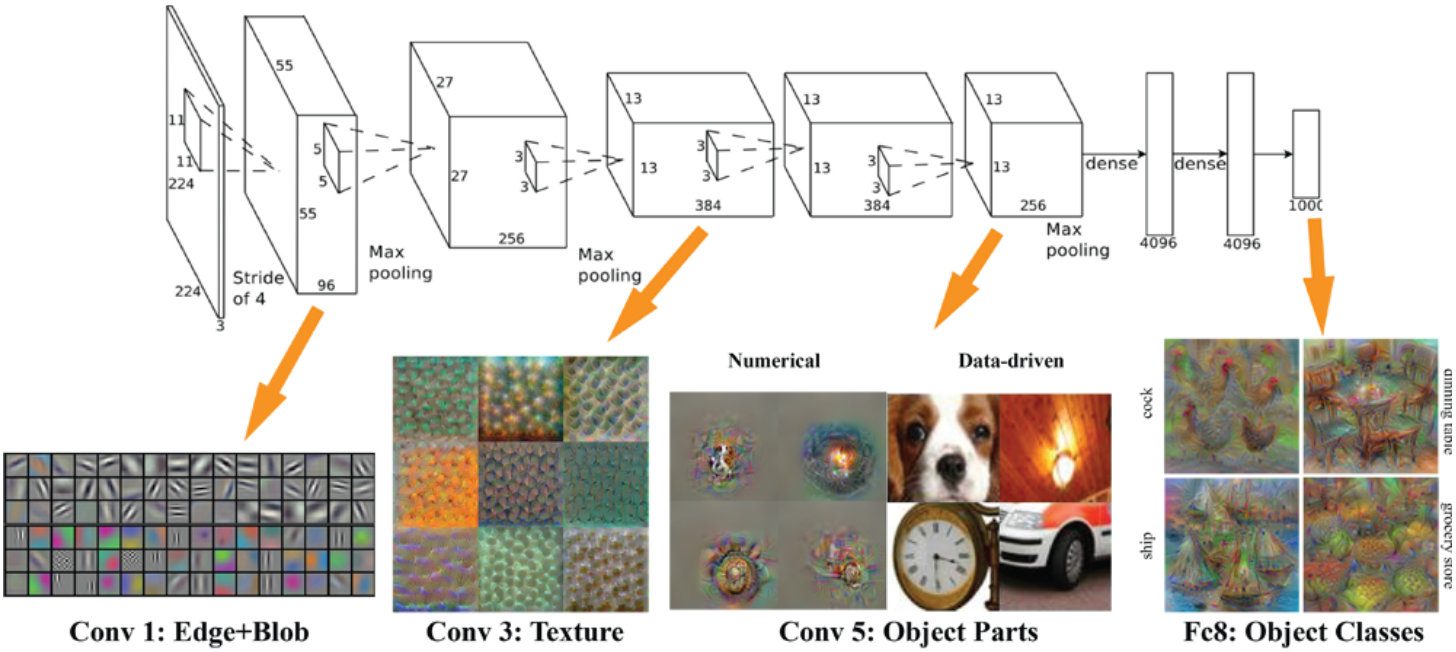


AlexNet / VGG-F network visualized by **mNeuron**.

Project 6: Deep Learning

Introduction to Computer Vision

Brief

- Due date: Sunday, December 2nd, 11:55pm
- Project materials including starter code, training/testing data and html writeup template: **proj6.zip** (83MB) updated 11/18/2018 at 12:30am.
- Handin: through **canvas.gatech.edu**
- Required files: README.txt, code/, html/, html/index.html. Make sure the submission size is <5MB. Clear the notebook output if necessary.

Overview

This project is an introduction to deep learning tools for computer vision. You will design and train deep convolutional networks for scene recognition using **PyTorch**, an open source deep learnig platform.

Remember **project 4: Scene recognition with bag of words**? You worked hard to design a bag of features representations that achieved 60 to 70% accuracy (most likely) on 15-way scene classification. You might have done the spatial pyramid extra credit and gotten up to nearly 80% accuracy. We're going to attack the same task with deep learning and get higher accuracy. Sort of -- training from scratch won't work quite as well as project 4, fine-tuning an existing network will work much better than project 4.

In Part 0 of the project you will train a simple network from scratch. You will run the code that is already provided (you do not need to write any code for this part) to train a simple network. In your report, report the performance of this simple network. This section will train a simple network that achieves only 30% to 35% accuracy (just slightly better than the tiny images baseline in project 4).

In Part 1, you will modify the dataloader used in Part 0 to include a few extra pre-processing steps. Pre-processing data allows the network to be more robust against small variations in the data. Examples of pre-processing transforms are jittering, flipping, and normalization (some of which are already called for you in the starter code). In Part 1 you will also modify the simple network by adding dropout, batch normalizations and more layers. These modifications might increase recognition accuracy to around 55%. Unfortunately, we only have 1,500 training examples so it doesn't seem possible to train a network from scratch which outperforms hand-crafted features (extra credit to anyone who proves us wrong).

For Part 2, you will instead *fine-tune* a pre-trained deep network to achieve more than 80% accuracy on the task. We will use the pretrained AlexNet network which was not trained to recognize scenes at all.

These different approaches (starting the training from scratch or fine-tuning) represent the most common approach to recognition problems in computer vision today -- train a deep network from scratch if you have enough data (it's not always obvious whether or not you do), and if you cannot then fine-tune a pre-trained network instead.

Starter Code Outline

The following is an outline of the student code:

- `create_datasets(...)`: Creates training and testing data loaders for 15 scene database. The data loaders include pre-processing steps.
- `create_part2_model(...)`: Modifies the passed in network for fine-tuning.
- `SimpleNet`: The simple network.
    - `__init__()` : Sets up and initializes the layers of the simple netowrk.
    - `forward()`: One forward pass of the network. You do not need to modify this function.

Part 0: Warm up

First, open the notebook and run Part 0. You trained your first deep network from scratch! As mentioned earlier, you do not need to write any code in this section--the dataloaders and network you used are already provided.

Part 1: Modifying the Dataloaders and the Simple Network

In Part 0 of the notebook you trained a deep network from scratch. Gone are the days of hand-designed features. Now we have end-to-end learning in which a highly non-linear representation is learned for our data to maximize our objective (in this case, 15-way classification accuracy). Instead of an anemic 70% accuracy we can now recognize scenes with... 35% accuracy. OK, that didn't work at all. What's going on?

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

```
def __init__(self, num_classes, droprate=0.5, rgb=False, verbose=False):
  in_channels = 3 if rgb else 1

  self.features = nn.Sequential(
    nn.Conv2d(in_channels=in_channels, out_channels=10, kernel_size=9,
      stride=1, padding=0, bias=False),
    nn.MaxPool2d(kernel_size=7, stride=7, padding=0),
    nn.ReLU(),
  )

  self.classifier = nn.Conv2d(in_channels=10, out_channels=num_classes,
    kernel_size=8, stride=1, padding=0) #fc

def forward(self, x):
  x = self.features(x)
  x = self.classifier(x)
  return x.squeeze()
```

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"). Since this is the case, there's not technically a mathematical difference from "convolutional" layers so we specify them in the same way in PyTorch. Note that in general, convolutional layers are NOT the same as fully connected layers.

Let's look at the first convolutional layer. The inputs to `nn.Conv2d(in_channels=1, out_channels=10, kernel_size=9, stride=1, padding=0, bias=False)` mean the filters have a 9x9 spatial resolution, span 1 filter depth (because the input images are grayscale), and that there are 10 filters. Its weights are the filters being learned. They are initialized with random numbers from a Gaussian distribution (`m.weight.data.normal_(0, 1)`). The network also learns a bias or constant offset to associate with the output of each filter. This is what `nn.init.constant(m.bias.data, 0)` initializes.

```
for name, m in self.named_modules():
  if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
    # initialize weights with randomly sampled numbers from a normal distribution
    m.weight.data.normal_(0, 1)
    m.weight.data.mul_(1e-2)
    if m.bias is not None:
      # initialize biases with zeros
      nn.init.constant(m.bias.data, 0)
  elif isinstance(m, nn.BatchNorm2d):
    pass
```

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 rectified linear (relu) in this project.

Note that the pool layer and relu layer have no *learned* parameters associated with them. We are hand-specifying their behavior entirely, so there are no weights to initialize as in the convolutional layers.

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* have 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 setting its inputs as `in_channels=10, out_channels=num_classes, kernel_size=8`. 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. 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 shown in the notebook. 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 regularization 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. Unfortunately, PyTorch only provides a very **basic visualization method** to help us figure this out. Here we set `verbose=True` to see the input size of the last convolution layer, `Input size to classifier is torch.Size([1, 10, 8, 8])`. We can also check the output shape of the last convolution layer, `Network output size is torch.Size([15])`. You might try other 3rd party packages to visulize better, e.g. **PyTorch Summary**.

If the last convolutional layer had a filter size of 6x6 that would lead to a output size of the final convolution layer of [3, 3, 15]. Those mean a 3x3 spatial resolution plus 15 depth (dimensions). We 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 artistry (read as: black magic) to design the right network and training strategy for optimal performance.

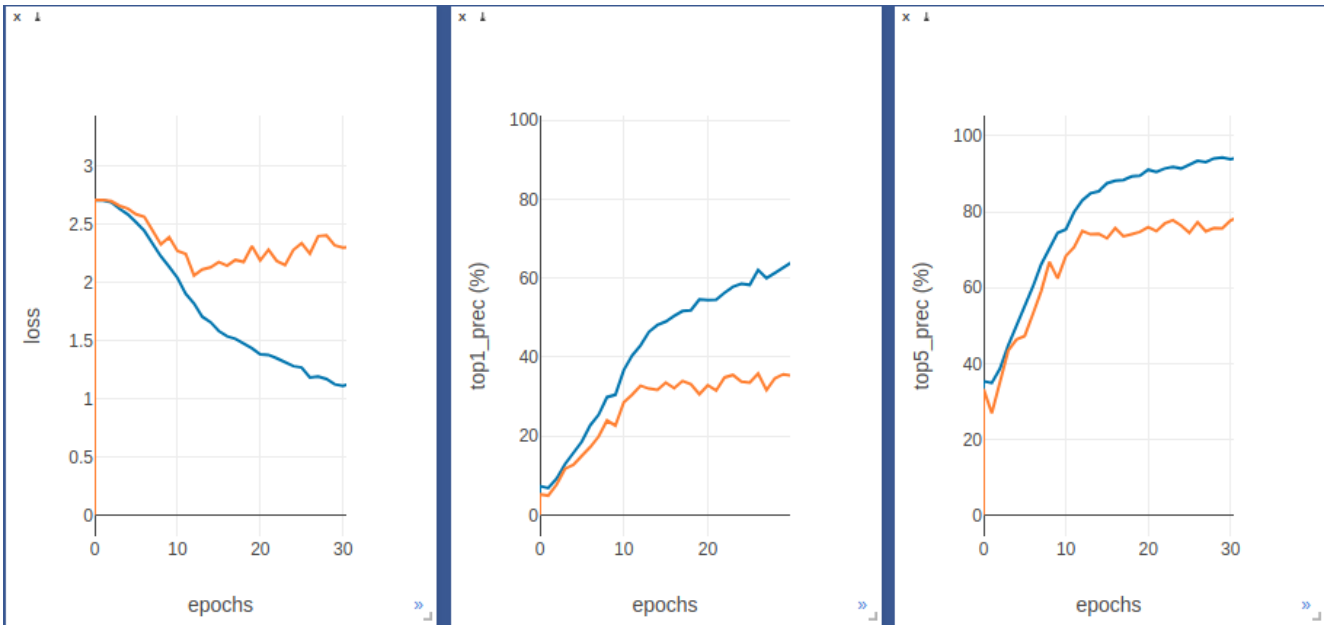
Luckily, we have a visulization figure from another package, MatConvNet, which was used in the previous version of this course (this is just for informational purposes, you do not need to worry about MatConvNet).

```
     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)|
```

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 thet 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) figures in Visdom (refer to your notebook for running instructions) should look like this:



Blue: training curve; Yellow: validation/test curve

We'll be studying these figures quite a bit during this project so it's important to understand what it shows.

The left pane shows the training loss (blue) and validation loss (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 for 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 line is the loss 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.

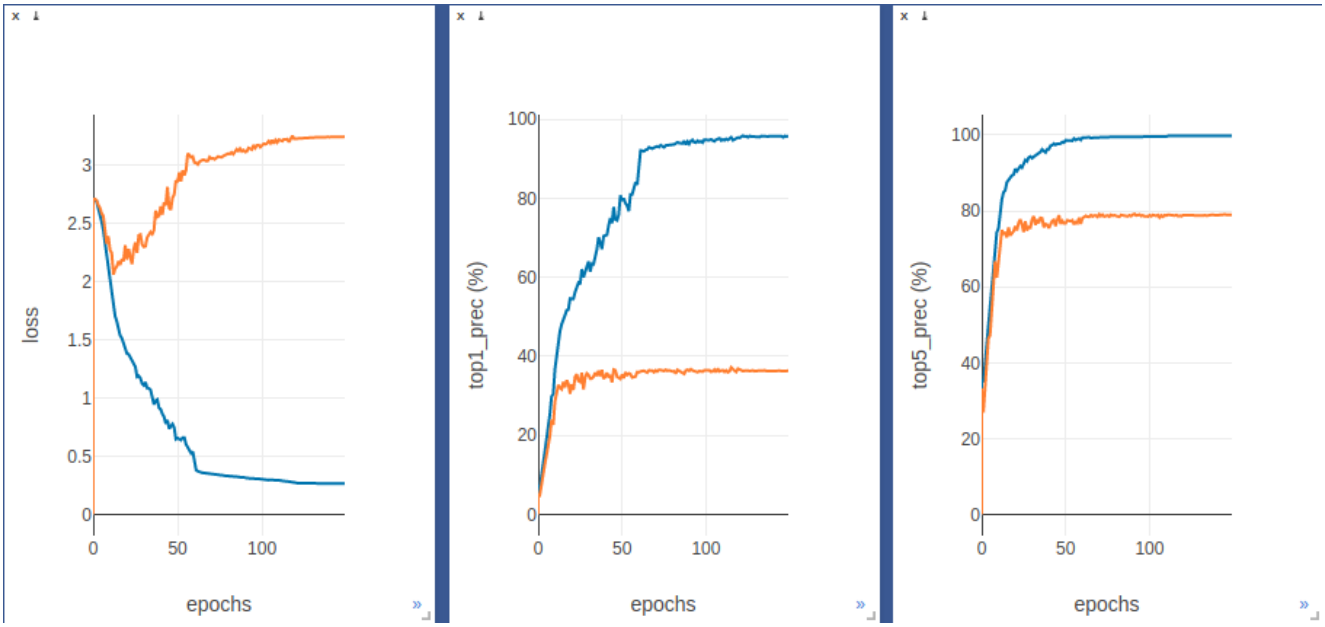
To be clear, accessing test accuracies in this particular project is merely for illustration purpose. You should not setup an automatic hyperparameter-tuning / model selection method using the val/test set (e.g. using the val/test set to do early stopping).

The middle pane shows the training and testing accuracy on the train and test (val) data sets across the same training epochs. It shows top 1 accuracy (top1 prec in the figures) -- how often the highest scoring guess is right. We're interested in top 1 accuracy, specifically the top 1 accuracy on the held out validation / test set.

The right pane shows top 5 accuracy -- how often all of the 5 highest scoring guesses are right. We're not as worried about this metric.

In this experiment, the training and test top 1 accuracy started out around 7% 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 right 7% of the time. As the training progressed and the network weights moved away from their random initialization, accuracy increased.

Let's look at the first 10 training epochs. During these epochs the training *and* validation accuracies are increasing, which is exactly what we want to see. Beyond that point the accuracy on the training dataset keeps increasing, but the validation accuracy does not. Our highest accuracy on the validation/test set is around 35%. 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:



Blue: training curve; Yellow: validation/test curve

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 accuracy with enough training data. 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 `base_lr = 1e-2` in `proj6.ipynb`) 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.

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 `create_datasets()` in `student_copy.py` each time it wants training or testing images. Modify `create_datasets()` to randomly flip some of the images (or entire batches) in the training dataset.

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 5% to 10% increase in accuracy (or drop in top 1 validation error) 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.

Useful function: `transforms.RandomHorizontalFlip`.

Problem 2: The images aren't zero-centered and variance-normalized.

One simple trick which can help a lot is to normalize the images by substracting their mean and then dividing their standard deviation. Modify `create_datasets()`. You should decide which mean or std you need to use for the training and test datasets. 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). After doing this you should see another 15% or so increase in accuracy. Most of this increase will show up in the first few iterations.

Useful functions: `transforms.Normalize`.

Problem 3: Our network isn't regularized.

If you train your network (especially for more than the default number of epochs) you'll see that the training accuracy can approach to 95% while the val accuracy hovers at 45% to 55%. 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:

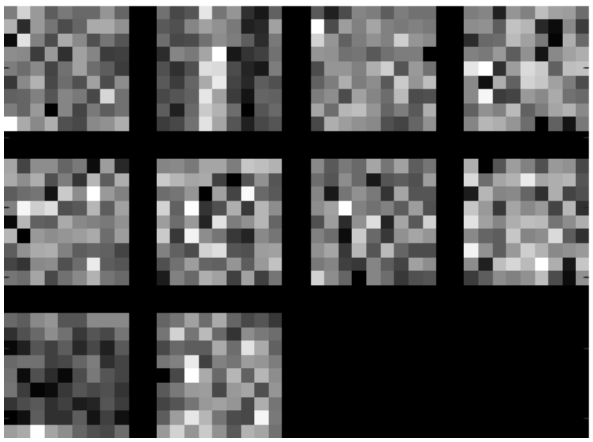
```
nn.Dropout(p=0.5)
```

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 time 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**.

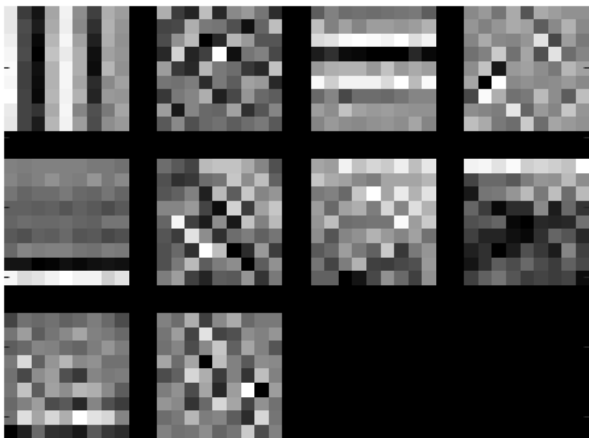
The dropout layer has only one free parameter (the dropout rate), which is the proportion of connections that are randomly deleted. The default of 0.5 should be fine. Insert a dropout layer between your convolutional layers. In particular, insert it directly before your last convolutional layer. Your 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 at least 55% validation accuracy. 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



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 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 zeroes 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. This architecture is reminiscent of hand-crafted features like the **gist scene descriptor** developed precisely for scene recoginition (on 8 scene categories which would later be incorporated into the 15 scene database). The gist descriptor actually works better than our learned feature. The gist descriptor with a non-linear classifier can achieve 74.7% accuracy on the 15 scene database.

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 PyTorch which contain 4 and 5 layers, respectively. AlexNet and VGG-F contain 8 layers, the **VGG "very deep" networks** contain 16 and 19 layers, and **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 the `SimpleNet` class. 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 `print('Network output size is', out.size())` in the notebook shows that your network's final layer (not counting the softmax) has a data size of 15, `Network output size is torch.Size([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.

Training deeper networks is tricky. The networks are slower to train and more sensitive to initialization and learning rate. For now, try to add one or two more blocks of "conv / pool / relu" and see if you can simply match your performance of the previous section.

Problem 5: Our "deep" network is slow to train and brittle.

You might have noticed that your deeper network doesn't seem to learn very reasonable filters in the first layer. It is harder for the gradients to pass from the last layer all the way to the first in a deeper architecture. Normalization can help. In particular, let's add a **batch normalization** layer after each convolutional layer except for the last. So if you have 4 total convolutional layers we will add 3 batch normalization layers. You can check out `nn.BatchNorm2d(num_features=...)`. You will also need to initialize the weights of the batch normalization layers. Set weight to 1 and bias to 0.

Batch normalization by itself won't necessarily increase accuracy, but it will allow you to use *much* higher learning rates. Try increasing your learning rate by a factor of 10 or even 100 and now you can rapidly explore and train different network architectures. Notice how the first layer filters start to show structure quickly with batch normalization.

We leave it up to you to determine the specifics of your deeper network: number of layers, number of filters in each layer, size of filters in each layer, padding, max-pooling, stride, dropout factor, etc. It is *not* required that your deeper network increases accuracy over the shallow network (but it can with the right hyperparameters). As long as you can achieve 50% test accuracy for some epoch 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 Part 1. Try to keep the part 1 training time under 10 minutes. You can achieve high accuracy in 100 epochs and in less in less than 10 minutes.

Additional optional improvements

Enjoy chasing higher accuracy? Here's some *optional* directions to investigate which might help improve your accuracy.

- If you look at PyTorch's ImageNet examples, you can see that the learning rate isn't constant during training. You can modify learning rate by changing `create_part1_trainer()`. This form of learning rate scheduling 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 tend to increase as you go up the network.
- 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.
- You can try alternate loss layers at the top of your network (e.g. `nn.MSELoss` and `nn.KLDivLoss`).
- You can train the AlexNet network from scratch on the 15 scene database. You can call `alexnet(pretrained=False)` to randomly initialized an AlexNet and train it just like your other networks. It works better than I would expect considering how little training data we have.

- The best accuracy James could achieve is 67.6% in 100 epochs without any data augmentation beyond mirroring and without changing the input resolution.

Part 2: fine-tuning a pre-trained deep network

One of the impressive things about the representations learned by deep convolutional networks is that they generalize surprisingly well to other recognition tasks (see **DeCAF** and the work of **Razavian et al.**). This is unexpected because these networks are discriminatively trained to perform well at a particular task, so one might expect their representations to "overfit" for that task. Perhaps they do, but they still often exceed the performance of hand-crafted features when used in a new domain.

But how do we use an existing deep network for a new recognition task? Take, for instance, **AlexNet**. **Strategy A**: The AlexNet network has 1000 units in the final layer corresponding to 1000 ImageNet categories. We could use those 1000 activations as a feature in place of a hand-crafted feature (such as a bag-of-features representation). You would train a classifier (typically a linear SVM) in that 1000-dimensional feature space. However, those activations are clearly very object-specific and may not generalize well to new recognition tasks. It is generally better to use the activations in slightly earlier layers of the network (e.g. the 4096 activations in "fc6" or "fc7"). You can often get away with sub-sampling those 4096 activations considerably (e.g. taking only the first 200 activations). **Strategy A** for using an existing deep network was extra credit for project 4 and several students achieved high accuracy (especially when using a deep network trained on the Places database, but that isn't so much a testament to generalization because it's the same task with more training data).

```
self.classifier = nn.Sequential(
    nn.Dropout(),
    nn.Linear(256 * 6 * 6, 4096), # fc6
    nn.ReLU(inplace=True),
    nn.Dropout(),
    nn.Linear(4096, 4096),        # fc7
    nn.ReLU(inplace=True),
    nn.Linear(4096, 1000),        # fc8
)
```

Alternatively, **Strategy B** is to *fine-tune* an existing network. In this scenario you take an existing network, replace the final layer (or more) with random weights, and train *the entire network* again with images and ground truth labels for your recognition task. You are effectively treating the pre-trained deep network as a better initialization than the random weights used when training from scratch. When you don't have enough training data to train a complex network from scratch (e.g. with the 15 scene database) this is an attractive option. Fine-tuning can work far better than **Strategy A** of taking the activations directly from an pre-trained CNN. For example, in **Lin et al's cross-view geolocalization work from CVPR 2015**, there wasn't enough data to train a deep network from scratch, but fine-tuning led to 4 times higher accuracy than using off-the-shelf networks directly.

For **Part 2** of this project you will fine-tune the AlexNet network to perform scene recognition.

The code for **Part 2** will largely follow the same outline or even use the same code as **Part 1**:

- `create_part2_model()` function will receive an AlexNet model and then *edit* the network rather than specifying the structure from scratch. You will edit AlexNet *while preserving* some of the learned weights.
- You need to make the following edits to the network: the final fc8 should be removed and specified again. The original fc8 had an input data depth of 4096 and an output data depth of 1000 (for 1000 ImageNet categories). We need the output depth to be 15, instead. The weights can be randomly initialized just like in Part 1.

With these issues addressed, you should see very high accuracy. Training will naturally be a bit slow because the network is far bigger than in Part 1. However, you don't need many training epochs. Four training epochs was enough to achieve 80% accuracy and it is possible to approach (or perhaps exceed) 88% test accuracy. Compare this with the 2010 state-of-the-art performance of 88.1% accuracy achieved by combining more than a dozen existing and new features with a non-linear SVM and multiple kernel learning in the **SUN Database paper**.

It isn't necessary to retrain the entire network to achieve high accuracy. You could retrain just the new fc8 layer by setting `backprop_depth` in the notebook. This basically implements **Strategy A** and would not be considered fine-tuning, but it is possible to do a strategy that falls in between. What if you only retrain the fully-connected layers? What if you prune the pre-trained network down to the convolutional layers and only add a single

fully-connected layer to dramatically reduce the number of parameters? There are many possible strategies to explore and you will receive full credit as long as you achieve 80% accuracy by starting from AlexNet. You can *additionally* experiment with fine-tuning other networks such as VGG or networks trained on the **Places database**, but be sure to report performance for and turn in code for fine-tuning AlexNet.

Write up

For this project, and all other projects, you must do a project report in HTML. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. Discuss any extra credit you did, and clearly show what contribution it had on the results (e.g. performance with and without each extra credit component). In your writeup, make sure to include the following:

- For Part 0, tell us what deep learning is, and why don't we need to engineer features to get high performing models. Also report the accuracy you got by training the model and the plots from the visdom server.
- For Part 1, explain jittering and normalization operations you performed on the datasets. Make sure to tell us why is it a good idea to use them, or what is their benefit. In addition, explain what is a sequential network and what do the feature extractor and the classifier parts do. Also explain specific layers one can use in a the feature extractor of a sequential network, mainly: Conv2d, MaxPool, Relu, Dropout, and BatchNorm. Make sure to talk about why do we use them. Finally, talk about the network structure or data transforms you used to exceed the 50% threshold. Tell us why you made the decisions you made, and make sure to include plots from the visdom server.
- For Part 2, talk about what is fine-tuning and how is it done. Tell us the benefits of fine-tuning and how you achieved at least the 80% threshold. Report the results you got by fine-tuning AlexNet, and include the graphs from the visdom server.
- If you implemented any extra credit, you should share and discuss the results in the writeup.

Do not include any code in your writeup.

Extra Credit

The max score for all students is 110.

For all extra credit, be sure to analyze on your web page whether your extra credit has improved classification accuracy. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit. Some ideas:

- up to 10 pts: Gather additional scene training data (e.g. from the **SUN database** or the **Places database**) and train a network from scratch. Report performance on those datasets and then use the learned networks for the 15 scene database with fine tuning.
- up to 10 pts: Try a completely different recognition task. For example, try to recognize **human object sketches** (download the .png files). Or try to predict **scene attributes**. The scene attributes are not one-vs-all (an image simultaneously has many attributes) so you'll need to configure your network accordingly. There are many other recognition data sets available to experiment with.
- up to 10 pts: Produce visualizations using your own code or methods such as **PyTorch CNN Visualization**
- up to 10 pts: 1 point for every percent accuracy over 65% when training from scratch on the 15 scene database. The highest accuracy James has gotten is 67%, so I expect this to require many bells and whistles such as extensive jittering of the training data, carefully tuned network structure, per-layer training rates, etc.
- up to 10 pts: 1 point for every percent accuracy over 90% when fine-tuning from AlexNet. You don't get extra credit for switching to another network (like one trained on the Places database). The challenge here is to adapt a very big network to a relatively small training set.
- up to 10 pts: Come up with your own idea to impress us. Chat with James or the TAs if you're not sure if an idea is worth pursuing.

Web-Publishing Results

All the results for each project will be put on the course website so that the students can see each other's results. In class we will highlight the best projects as determined by the professor and TAs. If you do not want your results published to the web, you can choose to opt out.

Handing in

This is very important as you will lose points if you do not follow instructions. Every time you do not follow instructions, you will lose 5 points. The folder you hand in must contain the following:

- README.txt - text file containing anything about the project that you want to tell the TAs
- code/ - directory containing your code and notebook for this assignment.
- html/ - directory containing all your html report for this assignment, including images (any images not under this directory won't be published).
- html/index.html - home page for your results

Hand in your project as a zip file through **canvas.gatech.edu**. Make sure the submission size is <5MB. Clear the notebook output if necessary.

Rubric

- +45 pts: Part 1: Build a convolutional network with pre-processing on the input data (jittering, normalization). Also add dropout regularization, batch normalization, and at least one additional convolutional layer which achieves at least 50% test accuracy (for any training epoch) on the 15 scene database. Your part 1 network should train under 10 minutes, without GPUs.
- +30 pts: Part 2: Fine-tune AlexNet to achieve at least 80% test accuracy on the 15 scene database. Your part 2 network should train under 10 minutes, without GPUs.
- +25 pts: Writeup with design decisions and evaluation. Refer back to the writeup section for more details.
- +10 pts: Extra credit (up to ten points)
- -5*n pts: Lose 5 points for every time you do not follow the instructions for the hand in format

Final Advice

- Deep Learning and and PyTorch are complicated. Use Piazza to get clarifications if needed. If you're confused, it's likely many other students are as well.

Credits

Project description and code by James Hays. Modified by Samarth Brahmbhatt, Jianan Gao and Dilara Soylu for Fall 2018.