Neuro 120 Homework 3: Vision and Deep Networks

In this homework you will investigate illusions in early vision, compare the representations in deep networks to those in the brain, and write your own backpropagation code to train a deep network. Starter code and data for the assignment is provided on Github. Work in groups of 2-4 students. You may use any resources or code you find online, but must properly attribute it. Submit all code you write, and the plots you produced to generate your answers.

1. **Early vision and convolutions.** A common model for retinal ganglion cell (RGC) responses is a difference of Gaussians (DoG) filter. This filter is also known as a center-surround filter, and has coefficients given by subtracting a Gaussian probability density function of larger standard deviation from one with smaller standard deviation. For processing images, these filter coefficients will be a two dimensional image patch $f(r,c) =$ where r and c are row and column indices. It has a central positive bump surrounded by an outer negative ring. Given an input image $I(r,c)$, the response r of a single retinal ganglion cell is given by element-wise multiplication of the filter and the input image followed by summation, $y = \sum_{r,c} f(r,c)I(r,c)$. However, there are many retinal ganglion cells with the same filter shape, but centered on different locations in the image. The response of all of these retinal ganglion cells can be computed through the convolution of the filter f with the image I, yielding the two-dimesional response $y(r,c)$. Here we will investigate several visual illusions that might arise from this type of processing. Starter code is provided in `earlyvision.m`. This script creates and plots three different images: a Mach bands image, which consists of a horizontal gradient; a checkerboard image, which consists of horizontal and vertical stripes; and a 'natural' image, which is a photo of some peppers.

   (a) Compute a difference of Gaussians filter of size 51 pixels by 51 pixels, positive standard deviation $\sigma_1 = 3$, and negative standard deviation $\sigma_2 = 7$. Plot the resulting filter coefficients using the `surf` command. There are many ways to do this in matlab but you may find the command `fspecial` to be the most convenient.

   (b) Convolve the Mach bands image with the filter from part A and plot the result. You will find the function `conv2` useful. One issue with convolutions is how to handle edge cases. Should RGCs be positioned only where their filters are completely inside the image? Or should they also be positioned in places where their filters extend outside the image, by padding the image with zeros? Here you can ignore these edge effects and only place RGCs where their filters are fully within the images. To do this, pass in 'valid' as the final argument to `conv2` (see the documentation of `conv2` for details). How might your result relate to the Mach band illusion?

   (c) The basic DoG filter is approximately mean zero, because it is the difference of probability density functions that both sum to one. This means that in constant regions of the input, the filter output will be zero regardless of whether the region is bright or dark. Try adding a small constant to the filter, to make it sensitive to the mean luminance. That is, try convolving the Mach bands image with a filter $f = DoG + .00001$. How might your result relate to the Mach band illusion?

   (d) Now consider the checkerboard image illusion, where dark spots appear at the intersections of the light stripes. Convolve the DoG filter of part A with the checkerboard image, and compare the resulting RGC output at the intersection vs in the middle of a stripe between intersections. How might your results relate to the checkerboard illusion?

   (e) When you look at the checkerboard illusion, you may notice that the dark spots are more visible in your peripheral vision than at your central fixation point. In the periphery, retinal ganglion cells have much broader filters than in your fovea. To model the behavior in your fovea, filter the image with a DoG of standard deviations $\sigma_1 = 0.5$ and $\sigma_2 = 1$. Plot this DoG filter using `surf`, and plot the resulting RGC response. How might your results relate to the vividness of the illusion in peripheral versus central vision?

   (f) Finally, try convolving the filter in part A with the natural image. Plot the response. Also try the smaller RGC filter from part E. How do the results compare? What features of the image are emphasized in the RGC response?

2. **Invariance and representational similarity in deep neural networks.** Here we will investigate an already-trained artificial deep neural network. The starter code in `deepnet_invariance.m` loads AlexNet, a breakthrough network that dramatically outperformed other object recognition algorithms when it was first introduced. The exact architecture and details are described in Krizhevsky, A., Hinton, G. E., & Sutskever, I. *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS 2012. To successfully load the network, you'll need to have matlab's Neural Network Toolbox installed, and download the additional AlexNet data file available at `webaddr`. The architecture of AlexNet consists of 25 layers (type `net.Layers` to see a list). Its input layer accepts 227×227 pixel images with three color channels. The other layers are `Convolution` layers, consisting of trainable weights that are applied everywhere in the image (as in the first homework problem); `ReLU` layers, which apply the rectified linear nonlinearity to each element of the response; `Normalization` layers which keep the norm of the activity roughly constant; `Pooling` layers, which compute the maximum response in a small part of the image to provide robustness to translations; `Dropout` layers which randomly zero out a fraction of neurons to help provide robustness and prevent overfitting; `Fully Connected` layers which include tunable connections between every neuron in the layer below to every neuron in the layer above; and a `Softmax` layer which normalizes all activities to sum to one and provides an estimate of the probability that the input image is from a specific class.

   (a) The first convolution layer contains 96 filters. Plot the weights of these filters (the first part of `deepnet_invariance.m` should do this for you). What types of neurons do you see?

   (b) Apply the network to the peppers image. Does it correctly classify the image? Investigate the activity of neurons in the first convolution layer. What are filters 3, 10, and 59 selective for?

   (c) Object recognition is challenging because nuisance variables like the position or rotation of an object can cause large changes between inputs at the pixel level. Load the images in `RDM_stimuli.mat`, which were used as part of an experiment by Kriegeskorte et al. *Matching Categorical Object Representations in Inferior Temporal Cortex of Man and Monkey.* Neuron, 2008. This will give you a variable `images` which contains 92 images used in that experiment. You will see that AlexNet is far from perfect–it's correct only about 60% of the time. More recent networks achieve much better error rates. Test the translation invariance of the network. In particular, take the German Shephard image (# 41), and compute the correlation between activations for the original image, and the image translated to the right by amounts in the range [0,180] pixels (you can subsample this range). Plot the correlation vs amount of translation for the `data`, `conv1`, `pool1`, `conv5`,`pool5`, `fc7`, and `prob` layers on the same plot. You may find the commands `imtranslate` and `corr` useful, as well as the `activation` command with the arguments 'OutputAs', 'columns'. Interpret your results: do lower or higher layers of the network exhibit more translation invariance? How do convolution layers affect translation invarince as compared to pooling layers?

   (d) Now test invariance to rotation. Rotate the German Shephard image in the range [0,180] degrees. You may find the command `imrotate` and the argument 'crop' useful. At what angle does AlexNet begin to misclassify this example? Based on your results, is AlexNet more resistant to translation or rotation?

   (e) A deep network may work well as a classifier, but do its representations align with those found in the brain? As a simple test of this, we will compare the *representational dissimilarity matrix* or RDM of various layers in AlexNet with that measured in IT cortex of monkeys (using electrode recordings) and humans (using the fMRI bold response). The RDM compares the correlation in neural population activity in response to different objects. Here, 92 objects $i = 1, \cdots, 92$ were presented and the resulting neural activity of N different neurons was recorded and formed into a vector $r_i \in R^N$ for each object. The RDM is a 92×92 matrix R where each element $R_{ij} = 1 - \rho_{ij}$ where ρ_{ij} is the correlation coefficient between population activity vectors r_i and r_j evoked by different objects. Hence a small entry in the RDM means that a very similar neural activity pattern was evoked in response to both images. See Kriegeskorte et al. *Matching Categorical Object Representations in Inferior Temporal Cortex of Man and Monkey.* Neuron, 2008 for further details of the experimental setup and dataset. Compute the RDM for AlexNet for the layers `data,conv1,pool1,pool2,relu3,relu4,relu5,pool5,fc6,fc7,fc8`. Compute the correlation coefficient between the AlexNet RDMs and the experimental RDMs `RDM1` and `RDM2` (eg, `corr(R(:),RDM1(:))`). Are the RDMs in intermediate layers of AlexNet more similar to the experimental RDMs than the raw data? Which layer of AlexNet best matches the experimental RDMs? You can use either RDM1 (electrode recordings in monkeys) or RDM2 (human fMRI responses). To calculate the RDM, note that the `corr` function automatically calculates the correlation matrix between all columns when given a matrix, such that $R = 1 - corr(F)$ if $F \in R^{N\times 92}$ is a matrix with the response of a layer of the network to each image in its columns.

3. **Backpropagation and deep learning.** In problem 2, we used a network that had already been trained on a massive dataset of labeled images. Here you will write your own training code and run it on a smaller scale problem: handwritten digit recognition. Starter code for this problem is available in `backprop.m`. The goal is to train a network to differentiate between handwritten 7's and 9's drawn from the MNIST dataset, a widely used proving ground for new learning algorithms before they are scaled up. This data is supplied in `mnist_sevens_nines.mat`, which will define the variables `X_train`, `y_train`, `X_test`, and `y_test`. There are 1800 training examples and 200 test examples. Sevens have been assigned target values of -1, while nines have been assigned target values of 1.

   We will implement a fully connected two hidden-layer network with ReLU nonlinearities, linear output neurons, and the mean squared error loss function. This network will have three matrices of tunable synapses $W_1 \in R^{N_{h_1}\times 784}, W_2 \in R^{N_{h_2}\times N_{h_1}}, W_3 \in R^{1\times N_{h_2}}$, where N_{h_1} and N_{h_2} are the number of hidden units in the first and second hidden layers respectively, and there is a single output neuron. These weights will be initialized as unit Gaussian random variables scaled by a `weight_scale` factor.

   Let's briefly review the mathematics of the backpropagation algorithm in this context. Given a dataset of P examples $\{x^\mu, y^\mu\}, \mu = 1, \cdots, P$, the loss function we wish to minimize is is

$$\mathcal{L}(W_3, W_2, W_1) = \frac{1}{2}\sum_{\mu=1}^{P} \|y^\mu - W_3 f(W_2 f(W_1 x^\mu)\|_2^2,$$

   where f is the ReLU nonlinearity applied elementwise.

   We wish to compute the gradient of this loss function with respect to each weight matrix. To do so, first, we compute a forward pass through the network to obtain its current prediction on an input. Given an input (column) vector $x \in R^{784}$, the first hidden layer activity is computed as $h_1 = f(W_1 x)$ where f is the ReLU nonlinearity is applied elementwise, and simply clips negative values to zero (i.e. $f(u) = max(u, 0)$). The second hidden layer activity is $h_2 = f(W_2 h_1)$. The output is $\hat{y} = W_3 h_2$ (which here is just a scalar). Now, we begin the backward pass through the network. We compare the network's output to the target value y to compute the error, $e = y - \hat{y}$, and this is the starting value of delta for backpropagation, $\delta_3 = e$. The backpropagation equations are

$$\delta_{l-1} = \left[W_l^T \delta_l\right] \circ f'(u_{l-1}),$$

   where $\circ$ denotes elementwise multiplication (`.*` in matlab). Here $f'(u_{l-1})$ is the derivative of the activation function evaluated at the activity level evoked by the example. In our case, the derivative of the ReLU function is simply 1 if the neuron's activity is greater than zero, and zero otherwise (we ignore the discontinuity at zero). Hence $f'(u_{l-1}) = \text{step}(h_{l-1})$ where $\text{step}(\cdot)$ is zero if its input is less than zero and 1 otherwise, and step is applied elementwise to a vector input. Finally, the gradient for the weights of layer l is

$$\frac{\partial \mathcal{L}}{\partial W_l} = -\delta_l h_{l-1}^T,$$

   where we define h_0 to be x.

   This procedure describes how to compute the gradient on one specific example $\{x, y\}$. To compute the full gradient, we just sum the gradients across examples. In fact, this summation can be handled as a matrix multiplication: all of the above steps work if x is a $784 \times P$ matrix and y is a $1 \times P$ row vector where P is the number of examples. Implementing the above steps will now yield matrices rather than vectors, and the final operation $\delta_{l+1} h_l^T$ will implement the summation across examples (you can also just do a simple for loop but it will be slower).

Finally, we update the weights by taking a small step in the negative gradient direction,

$$W \leftarrow W - \alpha \delta_l h_{l-1}^T,$$

where α is a learning rate.

(a) Implement gradient descent. Run 5000 iterations of gradient descent with a learning rate of $\alpha = 0.1/P$ (where P is the number of examples), and a `weight_scale` of 0.01. Plot the loss on the training examples and the loss on the testing examples over the course of learning, and the training/testing accuracy.

(b) How accurate a classifier does this yield? Is there substantial overfitting? Would stopping training early help the test accuracy?

(c) Now try a larger `weight_scale` of 0.1. Does this impact the final training accuracy? Does this impact the final testing accuracy? Would stopping training early help the test accuracy? Does a smaller or larger initial weight variance seem better for the purposes of generalization based on your findings?