



Red Green Blue

Inline Question: Comment below on what kinds of filters you see. Include your response in your submission [5pts]

Answer:

The first 16 5x5 filter maps (each having 3 channels) are visualized above. These filters look for lower level features in the images. The last filter is akin to the Emboss filter (having higher values on the upper traingle, and lower values on the lower traingle). There are also quite a few feature maps similar to Sobel (larger numbers on one half, lower numbers on the other). Other feature maps are arbitary, and capture certain spatial characteristics from the images. These are some of the recognisable filters from the 1st layer of the CNN model.

	 Learning Objective: In this problem, you will learn how to deeply understand how Convolutional Neural Networks work by implementing one. Provided Code: We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well. TODOs: you will implement a Convolutional Layer and a MaxPooling Layer to improve on your classification results in part 1. from lib.mlp.fully_conn import * from lib.mlp.layer_utils import * from lib.mlp.train import * from lib.mlp.train import * from lib.cnn.layer_utils import * from lib.cnn.layer_utils import * from lib.cnn.cnn models import *
	<pre>from lib.grad_check import * from lib.optim import * import numpy as np import matplotlib.pyplot as plt %matplotlib inline plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots plt.rcParams['image.interpolation'] = 'nearest' plt.rcParams['image.cmap'] = 'gray' # for auto-reloading external modules # see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython %load_ext autoreload %autoreload 2</pre>
	Loading the data (SVHN) Run the following code block to download SVHN dataset and load in the properly splitted SVHN data. The script <code>get_datasets.sh</code> use <code>wget</code> to download the SVHN dataset. If you have a trouble with executing <code>get_datasets.sh</code> , you can manually download the dataset and extract files. <code>!./get_datasets.sh</code> <code># !get_datasets.sh</code> for <code>windows users</code> <code>data = SVHN_data()</code> <code>for k, v in data.items():</code>
,	Name: {} Shape: {}".format(k, v.shape)) Name: data_train Shape: (70000, 32, 32, 3) Name: labels_train Shape: (70000,) Name: data_val Shape: (3257, 32, 32, 3) Name: labels_val Shape: (3257,) Name: data_test Shape: (26032, 32, 32, 3) Name: labels_test Shape: (26032,) Convolutional Neural Networks We will use convolutional neural networks to try to improve on the results from Problem 1. Convolutional layers make the assumption that local pixels are more important for prediction than far-away pixels. This allows us to form networks that are robust to small changes in
	Convolutional Layer Output size calculation [2pts] As you have learned, two important parameters of a convolutional layer are its stride and padding. To warm up, we will need to calculate the output size of a convolutional layer given its stride and padding. To do this, open the lib/cnn/layer_utils.py file and fill out the TODO section in the get_output_size function in the ConvLayer2D class. Implement your function so that it returns the correct size as indicated by the block below.
	<pre>input_image = np.zeros([32, 28, 28, 3]) # a stack of 32 28 by 28 rgb images in_channels = input_image.shape[-1] #must agree with the last dimension of the input image k_size = 4 n_filt = 16 conv_layer = ConvLayer2D(in_channels, k_size, n_filt, stride=2, padding=3) output_size = conv_layer.get_output_size(input_image.shape) print("Received {} and expected [32, 16, 16, 16]".format(output_size))</pre> Received [32, 16, 16, 16] and expected [32, 16, 16, 16]
	Convolutional Layer Forward Pass [5pts] Now, we will implement the forward pass of a convolutional layer. Fill in the TODO block in the forward function of the ConvLayer2D class. *reload_ext autoreload # Test the convolutional forward function input_image = np.linspace(-0.1, 0.4, num=1*8*8*1).reshape([1, 8, 8, 1]) # a single 8 by 8 grayscale image in_channels, k_size, n_filt = 1, 5, 2 weight_size = k_size*k_size*in_channels*n_filt
	<pre>bias_size = n_filt single_conv = ConvLayer2D(in_channels, k_size, n_filt, stride=1, padding=0, name="conv_test") w = np.linspace(-0.2, 0.2, num=weight_size).reshape(k_size, k_size, in_channels, n_filt) b = np.linspace(-0.3, 0.3, num=bias_size) single_conv.params[single_conv.w_name] = w single_conv.params[single_conv.b_name] = b out = single_conv.forward(input_image)</pre>
	<pre>print("Received output shape: {}, Expected output shape: (1, 4, 4, 2)".format(out.shape)) correct_out = np.array([[[[-0.03874312, 0.57000324], [-0.03955296, 0.57081309], [-0.04036281, 0.57162293], [-0.0417266, 0.57243278]], [[-0.0452219, 0.57648202], [-0.04603175, 0.57729187], [-0.04684159, 0.57810172], [-0.04765144, 0.57891156]], [[-0.05170068, 0.5829608],</pre>
	[-0.05251053, 0.58377065], [-0.05332038, 0.5845805], [-0.05413022, 0.58539035]], [[-0.05817946, 0.58943959], [-0.05898931, 0.59024943], [-0.05979916, 0.59105928], [-0.06060901, 0.59186913]]]]) # Compare your output with the above pre-computed ones. # The difference should not be larger than 1e-8 print ("Difference: ", rel_error(out, correct_out)) Received output shape: (1, 4, 4, 2), Expected output shape: (1, 4, 4, 2)
	Conv Layer Backward [5pts] Now complete the backward pass of a convolutional layer. Fill in the TODO block in the backward function of the ConvLayer2D class. Check you results with this code and expect differences of less than 1e-6. *reload_ext autoreload # Test the conv backward function img = np.random.randn(15, 8, 8, 3) w = np.random.randn(4, 4, 3, 12)
	<pre>b = np.random.randn(12) dout = np.random.randn(15, 4, 4, 12) single_conv = ConvLayer2D(input_channels=3, kernel_size=4, number_filters=12, stride=2, padding=1, name="consingle_conv.params[single_conv.w_name] = w single_conv.params[single_conv.b_name] = b dimg_num = eval_numerical_gradient_array(lambda x: single_conv.forward(img), img, dout) dw_num = eval_numerical_gradient_array(lambda w: single_conv.forward(img), w, dout) db_num = eval_numerical_gradient_array(lambda b: single_conv.forward(img), b, dout) out = single_conv.forward(img) dimg = single_conv.backward(dout)</pre>
	<pre>dw = single_conv.grads[single_conv.w_name] db = single_conv.grads[single_conv.b_name] # The error should be around 1e-8 print("dimg Error: ", rel_error(dimg_num, dimg)) # The errors should be around 1e-8 print("dw Error: ", rel_error(dw_num, dw)) print("db Error: ", rel_error(db_num, db)) # The shapes should be same print("dimg Shape: ", dimg.shape, img.shape) dimg Error: 7.637562229218304e-09 dw Error: 1.2802912969322507e-09 db Error: 2.0577980617311323e-10</pre>
	Max pooling Layer Now we will implement maxpooling layers, which can help to reduce the image size while preserving the overall structure of the image. Forward Pass max pooling [5pts] Fill out the TODO block in the forward function of the MaxPoolingLayer class. # Test the convolutional forward function
	<pre>input_image = np.linspace(-0.1, 0.4, num=64).reshape([1, 8, 8, 1]) # a single 8 by 8 grayscale image maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_test") out = maxpool.forward(input_image) print("Received output shape: {}, Expected output shape: (1, 3, 3, 1)".format(out.shape)) correct_out = np.array([[[[0.11428571], [0.13015873], [0.14603175]], [[0.24126984], [0.25714286], [0.27301587]]</pre>
	[0.27301587]], [[0.36825397], [0.38412698], [0.4]]]]) # Compare your output with the above pre-computed ones. # The difference should not be larger than 1e-8 print ("Difference: ", rel_error(out, correct_out)) Received output shape: (1, 3, 3, 1), Expected output shape: (1, 3, 3, 1) Difference: 1.8750000280978013e-08 Backward Pass Max pooling [5pts]
	Fill out the backward function in the MaxPoolingLayer class. img = np.random.randn(15, 8, 8, 3) dout = np.random.randn(15, 3, 3, 3) maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_test") dimg_num = eval_numerical_gradient_array(lambda x: maxpool.forward(img), img, dout) out = maxpool.forward(img) dimg = maxpool.backward(dout)
	# The error should be around 1e-8 print("dimg Error: ", rel_error(dimg_num, dimg)) # The shapes should be same print("dimg Shape: ", dimg.shape, img.shape) dimg Error: 1.8928936598382574e-11 dimg Shape: (15, 8, 8, 3) (15, 8, 8, 3) Test a Small Fully Connected Network [3pts] Please find the TestCNN class in lib/cnn/cnn_models.py . Again you only need to complete few lines of code in the TODO block. Please design a Convolutional> Maxpool> flatten> fc network where the shapes of parameters match the given shapes. Please
	<pre>insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively. Here you only modify the param_name part, the _w, and _b are automatically assigned during network setup. %reload_ext autoreload seed = 1234 np.random.seed(seed=seed) model = TestCNN() loss_func = cross_entropy() B, H, W, iC = 4, 8, 8, 8, 3 #batch, height, width, in_channels k = 3 #kernel size</pre>
	<pre>oC, Hi, O = 3, 27, 5 # out channels, Hidden Layer input, Output size std = 0.02 x = np.random.randn(B,H,W,iC) y = np.random.randint(O, size=B) print ("Testing initialization ") ###################################</pre>
	######################################
	b2 = np.linspace(-0.9, 0.1, num=0) ###################################
	<pre>scores = model.forward(feats) correct_scores = np.asarray([[-13.85107294, -11.52845818, -9.20584342, -6.88322866, -4.5606139], [-11.44514171, -10.21200524 , -8.97886878 , -7.74573231 , -6.51259584], [-9.03921048, -8.89555231 , -8.75189413 , -8.60823596, -8.46457778], [-6.63327925 , -7.57909937 , -8.52491949 , -9.4707396 , -10.41655972]]) scores_diff = np.sum(np.abs(scores - correct_scores)) assert_scores_diff < 1e-6, "Your implementation might be wrong!" print ("Passed!") print ("Testing the loss",) y = np.asarray([0, 2, 1, 4]) loss = loss_func.forward(scores, y) dLoss = loss_func.backward() correct_loss = 4.56046848799693</pre>
	<pre>assert abs(loss - correct_loss) < 1e-10, "Your implementation might be wrong!" print ("Passed!") print ("Testing the gradients (error should be no larger than 1e-6)") din = model.backward(dLoss) for layer in model.net.layers: if not layer.params: continue for name in sorted(layer.grads): f = lambda _: loss_func.forward(model.forward(feats), y) grad_num = eval_numerical_gradient(f, layer.params[name], verbose=False) print ('%s relative error: %.2e' % (name, rel_error(grad_num, layer.grads[name]))) Testing initialization</pre>
	Passed! Testing test-time forward pass Passed! Testing the loss Passed! Testing the gradients (error should be no larger than 1e-6) conv1_b relative error: 1.99e-09 conv1_w relative error: 3.24e-10 fc1_b relative error: 1.33e-10 fc1_w relative error: 3.89e-07 Training the Network [25pts]
	In this section, we defined a SmallConvolutionalNetwork class for you to fill in the TODO block in lib/cnn/cnn_models.py. Here please design a network with at most two convolutions and two maxpooling layers (you may use less). You can adjust the parameters for any layer, and include layers other than those listed above that you have implemented. You are also free to select any optimizer you have implemented (with any learning rate). Try to find a combination that is able to achieve 88% validation accuracy. # Arrange the data data_dict = { "data_train": (data["data_train"], data["labels_train"]), "data_val": (data["data_val"], data["labels_val"]),
n [10]: n [25]:	<pre>"data_test": (data["data_test"], data["labels_test"]) print("Data shape:", data_dict["data_train"][0].shape) print("Flattened data input size:", np.prod(data["data_train"].shape[1:])) print("Number of data classes:", max(data['labels_train']) + 1) Data shape: (70000, 32, 32, 3) Flattened data input size: 3072 Number of data classes: 10 %reload_ext autoreload</pre>
	<pre>seed = 123 np.random.seed(seed=seed) model = SmallConvolutionalNetwork() loss_f = cross_entropy() results = None ####################################</pre>
	<pre>batch_size = 100 epochs = 10 lr_decay = .999 lr_decay_every = 10 optimizer_rms = RMSProp(model.net, 1e-2) optimizer = SGD(model.net, 8e-2) ###################################</pre>
	<pre>opt_params, loss_hist, train_acc_hist, val_acc_hist = results (Iteration 1 / 7000) loss: 2.303081059595888 (Iteration 26 / 7000) loss: 2.2805942144178983 (Iteration 51 / 7000) loss: 2.200571775532671 (Iteration 76 / 7000) loss: 2.2110406366487916 (Iteration 101 / 7000) loss: 2.190221788486456 (Iteration 126 / 7000) loss: 2.20815095346584 (Iteration 151 / 7000) loss: 2.2359210897384165 (Iteration 176 / 7000) loss: 2.269811610846909 (Iteration 201 / 7000) loss: 2.2278256863342203 (Iteration 226 / 7000) loss: 2.217536422279763 (Iteration 251 / 7000) loss: 2.217862929801658 (Iteration 276 / 7000) loss: 2.2354030376611256</pre>
	(Iteration 301 / 7000) loss: 2.2857248179535627 (Iteration 326 / 7000) loss: 2.1737382508283907 (Iteration 351 / 7000) loss: 2.2206761547280416 (Iteration 376 / 7000) loss: 2.1848338460046106 (Iteration 401 / 7000) loss: 2.1984531301765475 (Iteration 426 / 7000) loss: 2.2576330191312945 (Iteration 451 / 7000) loss: 2.2282084695976474 (Iteration 476 / 7000) loss: 2.259346219696147 (Iteration 501 / 7000) loss: 2.2221840476587915 (Iteration 526 / 7000) loss: 2.236775120087029 (Iteration 576 / 7000) loss: 2.152931132145809 (Iteration 601 / 7000) loss: 2.1535610602543223
	(Iteration 626 / 7000) loss: 2.1482512652435917 (Iteration 651 / 7000) loss: 2.1550347129375735 (Iteration 676 / 7000) loss: 1.9951917069428255 (Epoch 1 / 10) Training Accuracy: 0.3840142857142857, Validation Accuracy: 0.3758059564015966 (Iteration 701 / 7000) loss: 1.8147499902156137 (Iteration 726 / 7000) loss: 1.719732303774698 (Iteration 751 / 7000) loss: 1.8591200208900647 (Iteration 776 / 7000) loss: 1.3446597228652906 (Iteration 801 / 7000) loss: 1.3185697607651559 (Iteration 826 / 7000) loss: 1.0720397349792772 (Iteration 851 / 7000) loss: 1.2237196678195779 (Iteration 976 / 7000) loss: 0.968542489617505 (Iteration 901 / 7000) loss: 1.0398282905213652 (Iteration 926 / 7000) loss: 1.0109870964405625
	(Iteration 926 / 7000) loss: 1.0109870964405625 (Iteration 951 / 7000) loss: 0.9659190040066923 (Iteration 1001 / 7000) loss: 0.9165748856933673 (Iteration 1026 / 7000) loss: 0.8554011963236001 (Iteration 1051 / 7000) loss: 0.6791535407761795 (Iteration 1076 / 7000) loss: 1.3015519952052008 (Iteration 1101 / 7000) loss: 0.8890751797433655 (Iteration 1126 / 7000) loss: 0.5017717804499813 (Iteration 1151 / 7000) loss: 0.7970508345219867 (Iteration 1201 / 7000) loss: 0.8078907818730534 (Iteration 1201 / 7000) loss: 0.6270056583575695 (Iteration 1226 / 7000) loss: 0.7826610987551973
	(Iteration 1276 / 7000) loss: 0.8998714656270611 (Iteration 1301 / 7000) loss: 0.7179145754029901 (Iteration 1326 / 7000) loss: 0.6588310257259172 (Iteration 1376 / 7000) loss: 0.7423826643867764 (Iteration 1376 / 7000) loss: 0.9298051777829918 (Epoch 2 / 10) Training Accuracy: 0.8285428571428571, Validation Accuracy: 0.8203868590727663 (Iteration 1401 / 7000) loss: 0.8721932606928543 (Iteration 1426 / 7000) loss: 0.9325982256521834 (Iteration 1451 / 7000) loss: 0.5490903126276906 (Iteration 1476 / 7000) loss: 0.7514261740774626 (Iteration 1501 / 7000) loss: 1.1836911022582008 (Iteration 1551 / 7000) loss: 0.549679174226565 (Iteration 1576 / 7000) loss: 0.7669239613095016
	(Iteration 1601 / 7000) loss: 0.7219554306936874 (Iteration 1626 / 7000) loss: 0.6528383516778853 (Iteration 1651 / 7000) loss: 0.6053171818552048 (Iteration 1676 / 7000) loss: 0.7271232563564854 (Iteration 1701 / 7000) loss: 0.7539949245363641 (Iteration 1726 / 7000) loss: 0.549860169926417 (Iteration 1751 / 7000) loss: 0.7169827430846055 (Iteration 1776 / 7000) loss: 0.6709361983076554 (Iteration 1801 / 7000) loss: 0.5194899186040396 (Iteration 1826 / 7000) loss: 0.6905292783357992 (Iteration 1851 / 7000) loss: 0.744743066278433 (Iteration 1876 / 7000) loss: 0.6051477426953162 (Iteration 1901 / 7000) loss: 0.4996035267347959
	(Iteration 1926 / 7000) loss: 0.5525588501104469 (Iteration 1951 / 7000) loss: 0.49892075247364703 (Iteration 1976 / 7000) loss: 0.6218438587338445 (Iteration 2001 / 7000) loss: 0.6385646971637398 (Iteration 2026 / 7000) loss: 0.6210294082026253 (Iteration 2051 / 7000) loss: 0.5580696571477061 (Iteration 2076 / 7000) loss: 0.9036588189737362 (Epoch 3 / 10) Training Accuracy: 0.8545142857142857, Validation Accuracy: 0.8440282468529321 (Iteration 2101 / 7000) loss: 0.6125410724915614 (Iteration 2126 / 7000) loss: 0.6530451199294884 (Iteration 2151 / 7000) loss: 0.6654972349604545 (Iteration 2201 / 7000) loss: 0.5059808318550864 (Iteration 2226 / 7000) loss: 0.6383227440133526
	(Iteration 2251 / 7000) loss: 0.6739976031120495 (Iteration 2276 / 7000) loss: 0.7456347114271111 (Iteration 2301 / 7000) loss: 0.49129075348318596 (Iteration 2326 / 7000) loss: 0.619050214772351 (Iteration 2351 / 7000) loss: 0.5827992001828832 (Iteration 2376 / 7000) loss: 0.5940286932775066 (Iteration 2401 / 7000) loss: 0.5872374027741047 (Iteration 2426 / 7000) loss: 0.6161599178793282 (Iteration 2476 / 7000) loss: 0.4411723616626623 (Iteration 2501 / 7000) loss: 0.4709408181353949 (Iteration 2526 / 7000) loss: 0.5619242003969369 (Iteration 2551 / 7000) loss: 0.7571816922295127
	(Iteration 2576 / 7000) loss: 0.44055593532319237 (Iteration 2601 / 7000) loss: 0.8381262177377872 (Iteration 2626 / 7000) loss: 0.7113128941488261 (Iteration 2651 / 7000) loss: 0.43346174917908514 (Iteration 2676 / 7000) loss: 0.5275713186514465 (Iteration 2701 / 7000) loss: 0.5247172563465273 (Iteration 2751 / 7000) loss: 0.7355501491930602 (Iteration 2776 / 7000) loss: 0.7173985839586966 (Epoch 4 / 10) Training Accuracy: 0.8641571428571428, Validation Accuracy: 0.8578446423088733 (Iteration 2801 / 7000) loss: 0.5714023514239714 (Iteration 2826 / 7000) loss: 0.5609925612404699 (Iteration 2851 / 7000) loss: 0.2952830460055606
	(Iteration 2876 / 7000) loss: 0.8275057985475479 (Iteration 2901 / 7000) loss: 0.7003610107739323 (Iteration 2926 / 7000) loss: 0.39618682772421693 (Iteration 2951 / 7000) loss: 0.4883029617761203 (Iteration 2976 / 7000) loss: 0.34115511617038097 (Iteration 3001 / 7000) loss: 0.6807840473833576 (Iteration 3026 / 7000) loss: 0.7955964967591957 (Iteration 3051 / 7000) loss: 0.6807169053271775 (Iteration 3076 / 7000) loss: 0.5311465168488438 (Iteration 3101 / 7000) loss: 0.6008811296223062 (Iteration 3126 / 7000) loss: 0.5677132880273276 (Iteration 3176 / 7000) loss: 0.5418246797104842 (Iteration 3201 / 7000) loss: 0.55153002021757364
	(Iteration 3201 / 7000) loss: 0.5153002021757364 (Iteration 3226 / 7000) loss: 0.44896898445800476 (Iteration 3251 / 7000) loss: 0.6438467932788194 (Iteration 3276 / 7000) loss: 0.6108185034727786 (Iteration 3301 / 7000) loss: 0.5444955975624218 (Iteration 3326 / 7000) loss: 0.5263176822607107 (Iteration 3351 / 7000) loss: 0.5548728920810951 (Iteration 3376 / 7000) loss: 0.5850815237427252 (Iteration 3401 / 7000) loss: 0.7222866174616565 (Iteration 3426 / 7000) loss: 0.6391347710368045 (Iteration 3476 / 7000) loss: 0.7507475320478008 (Epoch 5 / 10) Training Accuracy: 0.8733285714285715, Validation Accuracy: 0.8710469757445501 (Iteration 3501 / 7000) loss: 0.2909835610535152
	(Iteration 3526 / 7000) loss: 0.5361529639817113 (Iteration 3551 / 7000) loss: 0.48010107382392037 (Iteration 3576 / 7000) loss: 0.3574942383190266 (Iteration 3601 / 7000) loss: 0.5058604217973616 (Iteration 3626 / 7000) loss: 0.5247782243540617 (Iteration 3651 / 7000) loss: 0.5801157259477513 (Iteration 3676 / 7000) loss: 0.4190059664602437 (Iteration 3701 / 7000) loss: 0.5343070762758156 (Iteration 3726 / 7000) loss: 0.5442838321767856 (Iteration 3776 / 7000) loss: 0.572974526477804 (Iteration 3801 / 7000) loss: 0.4801984965371545 (Iteration 3826 / 7000) loss: 0.5183667146255039
	(Iteration 3851 / 7000) loss: 0.6055379624653006 (Iteration 3876 / 7000) loss: 0.6577640478333077 (Iteration 3901 / 7000) loss: 0.3942038390763621 (Iteration 3926 / 7000) loss: 0.39485194538532786 (Iteration 3976 / 7000) loss: 0.3907288507553696 (Iteration 3976 / 7000) loss: 0.7400237183675649 (Iteration 4001 / 7000) loss: 0.4302892176691851 (Iteration 4026 / 7000) loss: 0.6746132768879782 (Iteration 4051 / 7000) loss: 0.5392347398261547 (Iteration 4076 / 7000) loss: 0.4742694417630662 (Iteration 4101 / 7000) loss: 0.48812666576522223 (Iteration 4126 / 7000) loss: 0.8489458832038923 (Iteration 4151 / 7000) loss: 0.49149002156518096 (Iteration 4176 / 7000) loss: 0.5718364139071933
	(Epoch 6 / 10) Training Accuracy: 0.8779285714285714, Validation Accuracy: 0.8731961928154743 (Iteration 4201 / 7000) loss: 0.5280526234162068 (Iteration 4226 / 7000) loss: 0.42854591918673024 (Iteration 4251 / 7000) loss: 0.5863946525296344 (Iteration 4276 / 7000) loss: 0.2880977310642385 (Iteration 4301 / 7000) loss: 0.32020891366703885 (Iteration 4326 / 7000) loss: 0.5377722712082208 (Iteration 4351 / 7000) loss: 0.4040714424397808 (Iteration 4376 / 7000) loss: 0.27599492828207917 (Iteration 4401 / 7000) loss: 0.4075569322987711 (Iteration 4426 / 7000) loss: 0.44878076057273186 (Iteration 4451 / 7000) loss: 0.5934792199505905 (Iteration 4476 / 7000) loss: 0.5097758831049217 (Iteration 4501 / 7000) loss: 0.5636820104226011
	(Iteration 4501 / 7000) loss: 0.6353188259875631 (Iteration 4551 / 7000) loss: 0.5965130676365049 (Iteration 4576 / 7000) loss: 0.6019581051196375 (Iteration 4601 / 7000) loss: 0.6649455785843138 (Iteration 4651 / 7000) loss: 0.43449854210919014 (Iteration 4676 / 7000) loss: 0.3205482147019545 (Iteration 4701 / 7000) loss: 0.8332919981453546 (Iteration 4701 / 7000) loss: 0.4273245641922696 (Iteration 4751 / 7000) loss: 0.49515539577302164 (Iteration 4801 / 7000) loss: 0.5212658096549196 (Iteration 4826 / 7000) loss: 0.5646853346664767
	(Iteration 5151 / 7000) loss: 0.49895985648978725 (Iteration 5176 / 7000) loss: 0.4810595745819856 (Iteration 5201 / 7000) loss: 0.5807104771685772 (Iteration 5226 / 7000) loss: 0.4052912184680988 (Iteration 5276 / 7000) loss: 0.5897685781511062 (Iteration 5301 / 7000) loss: 0.5923355885191982 (Iteration 5302 / 7000) loss: 0.4502465124360797 (Iteration 5326 / 7000) loss: 0.37722801713157744 (Iteration 5376 / 7000) loss: 0.3369269225524848 (Iteration 5401 / 7000) loss: 0.5791940844134152 (Iteration 5401 / 7000) loss: 0.522413326066409 (Iteration 5451 / 7000) loss: 0.514021903173735 (Iteration 5476 / 7000) loss: 0.5006096067750788
	(Iteration 5501 / 7000) loss: 0.5296049420028593 (Iteration 5526 / 7000) loss: 0.5342727987163218 (Iteration 5551 / 7000) loss: 0.5794662474016365 (Iteration 5576 / 7000) loss: 0.44241298753964364 (Epoch 8 / 10) Training Accuracy: 0.8838142857142857, Validation Accuracy: 0.8738102548357384 (Iteration 5601 / 7000) loss: 0.31691573057504685 (Iteration 5626 / 7000) loss: 0.4814415336070397 (Iteration 5651 / 7000) loss: 0.6111359015276374 (Iteration 5706 / 7000) loss: 0.4588470283047952 (Iteration 5726 / 7000) loss: 0.48350133115579735 (Iteration 5751 / 7000) loss: 0.5218582754372022 (Iteration 5776 / 7000) loss: 0.5619785126872817 (Iteration 5801 / 7000) loss: 0.49656590471904977
	(Iteration 5801 / 7000) loss: 0.49656590471904977 (Iteration 5826 / 7000) loss: 0.4854985361035789 (Iteration 5851 / 7000) loss: 0.5997418589407723 (Iteration 5876 / 7000) loss: 0.3996642715893655 (Iteration 5901 / 7000) loss: 0.4997301784171883 (Iteration 5926 / 7000) loss: 0.621104133421247 (Iteration 5951 / 7000) loss: 0.6192735031650438 (Iteration 5976 / 7000) loss: 0.5503696842002096 (Iteration 6001 / 7000) loss: 0.489611546423125 (Iteration 6026 / 7000) loss: 0.4360831515762919 (Iteration 6051 / 7000) loss: 0.6793028261140259 (Iteration 6076 / 7000) loss: 0.3512608789994598 (Iteration 6101 / 7000) loss: 0.8161083615011188 (Iteration 6126 / 7000) loss: 0.5650119359487775
	(Iteration 6151 / 7000) loss: 0.46946050102143866 (Iteration 6176 / 7000) loss: 0.3791966062090922 (Iteration 6201 / 7000) loss: 0.5528851483730503 (Iteration 6226 / 7000) loss: 0.618389208963339 (Iteration 6251 / 7000) loss: 0.4506057398594228 (Iteration 6276 / 7000) loss: 0.4633755669579347 (Epoch 9 / 10) Training Accuracy: 0.8866571428571428, Validation Accuracy: 0.8753454098863985 (Iteration 6301 / 7000) loss: 0.5007020498668742 (Iteration 6351 / 7000) loss: 0.765152359479816 (Iteration 6376 / 7000) loss: 0.5175101777342029 (Iteration 6401 / 7000) loss: 0.35009249804015197 (Iteration 6401 / 7000) loss: 0.39038383621176836 (Iteration 6426 / 7000) loss: 0.5335756277266468
	(Iteration 6451 / 7000) loss: 0.330715789034037 (Iteration 6476 / 7000) loss: 0.5293366798325978 (Iteration 6501 / 7000) loss: 0.5801907807505712 (Iteration 6526 / 7000) loss: 0.375581048097592 (Iteration 6551 / 7000) loss: 0.3853083170592769 (Iteration 6576 / 7000) loss: 0.559824302979582 (Iteration 6601 / 7000) loss: 0.456977606683737 (Iteration 6626 / 7000) loss: 0.49864975293766184 (Iteration 6651 / 7000) loss: 0.4813002805137146 (Iteration 6676 / 7000) loss: 0.6743242369394794 (Iteration 6701 / 7000) loss: 0.38051539836828796 (Iteration 6726 / 7000) loss: 0.405247432716685 (Iteration 6751 / 7000) loss: 0.4676219652801494 (Iteration 6776 / 7000) loss: 0.5776256395594482
n [27]:	(Iteration 6801 / 7000) loss: 0.41124451267601636 (Iteration 6826 / 7000) loss: 0.5927606544308551 (Iteration 6851 / 7000) loss: 0.4929468321266624 (Iteration 6876 / 7000) loss: 0.4725039595556984 (Iteration 6901 / 7000) loss: 0.5802958445284332 (Iteration 6901 / 7000) loss: 0.5061918849706769 (Iteration 6951 / 7000) loss: 0.5302878807333025 (Iteration 6976 / 7000) loss: 0.7363468374632223 (Epoch 10 / 10) Training Accuracy: 0.8889285714285714, Validation Accuracy: 0.8741172858458705 #Validation Accuracy Check - 87.4% # CONV->MAXPOOL->CONV->MAXPOOL->FLATTEN->LRELU->DROPOUT->FULLY CONNECTED" modelo = SmallConvolutionalNetwork() modelo.net.load(opt params)
	<pre>val_acc = compute_acc(modelo, data["data_val"], data["labels_val"]) print ("Validation Accuracy: {}%".format(val_acc*100)) test_acc = compute_acc(modelo, data["data_test"], data["labels_test"]) print ("Testing Accuracy: {}%".format(test_acc*100)) Loading Params: conv1_w Shape: (5, 5, 3, 16) Loading Params: conv1_b Shape: (16,) Loading Params: conv2_w Shape: (3, 3, 16, 32) Loading Params: conv2_b Shape: (32,) Loading Params: fc1_w Shape: (1152, 10) Loading Params: fc1_b Shape: (10,) Validation Accuracy: 87.41172858458704% Testing Accuracy: 86.70482483097726%</pre>
n [26]:	Run the code below to generate the training plots. %reload_ext autoreload opt_params, loss_hist, train_acc_hist, val_acc_hist = results # Plot the learning curves plt.subplot(2, 1, 1) plt.title('Training loss') loss_hist_ = loss_hist[1::100] # sparse the curve a bit plt.plot(loss_hist_, '-o') plt.xlabel('Iteration')
	<pre>plt.subplot(2, 1, 2) plt.title('Accuracy') plt.plot(train_acc_hist, '-o', label='Training') plt.plot(val_acc_hist, '-o', label='Validation') plt.xlabel('Epoch') plt.legend(loc='lower right') plt.gcf().set_size_inches(15, 12) plt.show()</pre> Training loss
	2.25 - 2.00 - 1.75 - 1.50 - 1.25 - 1.00 - 0.75 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 1.00 - 0.50 - 0.50 - 1.00 - 0.50 - 0.
	0.50 -
	0.7 - 0.6 - 0.5 - 0.4 - 0.6 - 0.5 - Epoch Training Validation
	Visualizing Layers [5pts] An interesting finding from early research in convolutional networks was that the learned convolutions resembled filters used for things like
	<pre>edge detection. Complete the code below to visualize the filters in the first convolutional layer of your best model. import matplotlib.pyplot as plt filters = model.net.get_params("conv1_w") biases = model.net.get_params("conv1_b") f_min, f_max = filters.min(), filters.max()</pre>
,	<pre>import matplotlib.pyplot as plt filters = model.net.get_params("conv1_w") biases = model.net.get_params("conv1_b")</pre>
n [28]:	<pre>import matplotlib.pyplot as plt filters = model.net.get_params("conv1_w") biases = model.net.get_params("conv1_b") f_min, f_max = filters.min(), filters.max() filters = (filters - f_min) / (f_max - f_min) im_array = None nrows, ncols = None, None ###################################</pre>
n [28]:	<pre>import matplotlib.pyplot as plt filters = model.net.get_params("conv1_w") biases = model.net.get_params("conv1_b") f_min, f_max = filters.min(), filters.max() filters = (filters - f_min) / (f_max - f_min) im_array = None nrows, ncols = None, None ###################################</pre>
n [28]:	<pre>import matplotlib.pyplot as plt filters = model.net.get_params("conv1_w") biases = model.net.get_params("conv1_b") f_min, f_max = filters.min(), filters.max() filters = (filters - f_min) / (f_max - f_min) im_array = None nrows, ncols = None, None ###################################</pre>