

Lab Assignment 4

Convolutional Neural Networks

Eleftherios Karamoulas - S3261859
Panagiotis Tzafos - S3302148

March 27, 2017

2 Theory questions

1. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting(CS231n lecture). In practice, its use is to downsample its input, reducing the amount of resources that the network needs in order to perform the computations by reducing the dimensions of the input using a filter, a stride and an elementwise activation function. Also, after the downsampling it is obvious that the network will have less parameters. As a result, it will be able to generalize better in new situations and avoid overfitting(small training set error, large error for new examples).
2. Weight sharing is a very important feature, as it can dramatically reduce the number of weights(less computations). The idea behind this is that the number of unique sets of weights can be equal to the number of filters that are going to be used, as we are about use these to every different region of the image we are scanning. This is when the network is interested to examine if a feature exists in the image indepentantly by its position. It can be applied because of the fact that, assuming that a single depth slice(filter) weight configuration is associated with a single feature that our network is looking for(edges, circles etc.), in every single one spatial region that the filter checks, it responsible of tracing these specific features(Not useful when the network has to learn in different positions of the image different features, e.g. a training set of faces, that are centralized).
3. The time is less in ReLu activation function compared to the sigmoid and the tanh activation functions, as it does not involve expensive operations(exponentials, etc.), as it can be implemented by simply thresholding a matrix of activations at zero. The results are better because of the properties that ReLu function provides, like sparse representation(the more activations are ≤ 0 the more sparse is our model and this is generally more beneficial than dense representations that sigmoid and tanh provide) and ReLu function non-saturating form(sigmoid and tanh functions are vanishing gradient, as the absolute value of x in $\alpha = Wx + b$ increases the gradient of sigmoid becomes increasingly small compared to ReLu that the gradient is constant.)
4. (a) From the given data and the formula to compute the output we have that $W = 12, F = 3, P = 0, S = 1$, so output $= \frac{W-F+2*P}{S} + 1 = \frac{12-3+2*0}{1} + 1 = 10$ and because we have 3 filters the total neurons will be $10 \times 10 \times 3 = 300$. The total number of parameters will be 300×1 weights for the full depth(grayscale) of the input image $+ 3$ for the 3 biases of each filters $= 303$.

(b) The input layer is our image and if we assume that each pixel is one neuron, our input layer consists of $12 \times 12 = 144$ neurons and each of them will be fully connected to

our neurons in the hidden layer which consists of 300 neurons. As a result the total connections between input and hidden layer will be $144 \times 300 = 43200$ weights + 300 biases = 43500. It is obvious that the number of parameters of the fully-connected layer compared with a convolutional layer is much bigger, and that makes convolutional layer more efficient.

5. This approach to solve the car decision problem is questionable because even though we can use CNN's and get a network that can decide between cars an average solution, the features that characterise the car can be subjective for different customers depending on their needs and comforts.

4 Convolutional Layer

```

1 function convolvedFeatures = cnnConvolve(filterDim, numFilters, images, W, b)
2 %cnnConvolve Returns the convolution of the features given by W and b with
3 %the given images
4 %
5 % Parameters:
6 % filterDim - filter (feature) dimension
7 % numFilters - number of feature maps
8 % images - large images to convolve with, matrix in the form
9 %           images(r, c, image number)
10 % W, b - W, b for features from the sparse autoencoder
11 %        W is of shape (filterDim,filterDim,numFilters)
12 %        b is of shape (numFilters,1)
13 %
14 % Returns:
15 % convolvedFeatures - matrix of convolved features in the form
16 %                    convolvedFeatures(imageRow, imageCol, featureNum, imageNum)
17
18 numImages = size(images, 3);
19 imageDim = size(images, 1);
20 convDim = imageDim - filterDim + 1;
21 convolvedFeatures = zeros(convDim, convDim, numFilters, numImages);
22
23 % Instructions:
24 % Convolve every filter with every image here to produce the
25 % (imageDim - filterDim + 1) x (imageDim - filterDim + 1) x numFeatures x numImages
26 % matrix convolvedFeatures, such that
27 % convolvedFeatures(imageRow, imageCol, featureNum, imageNum) is the
28 % value of the convolved featureNum feature for the imageNum image over
29 % the region (imageRow, imageCol) to (imageRow + filterDim - 1, imageCol + filterDim -
30 % 1)
31 %
32 % Expected running times:
33 % Convolving with 100 images should take less than 30 seconds
34 % Convolving with 5000 images should take around 2 minutes
35 % (So to save time when testing, you should convolve with less images, as
36 % described earlier)
37 %%% Add code here
38 for i=1:numImages

```

```

39     im = images(:,:,i);
40     for j=1:numFilters
41         cw = rot90(W(:,:,j),2);
42         cf = conv2(im,cw,'valid') + b(j);
43         convolvedFeatures(:,:,j,i) = 1./(1+exp(-cf));
44     end
45 end
46 end

```

Listing 1: cnnConvolve.m

5 Mean Pooling Layer

```

1 function pooledFeatures = cnnPool(poolDim, convolvedFeatures)
2 %cnnPool Pools the given convolved features
3 %
4 % Parameters:
5 % poolDim - dimension of pooling region
6 % convolvedFeatures - convolved features to pool (as given by cnnConvolve)
7 %                     convolvedFeatures(imageRow, imageCol, featureNum, imageNum)
8 %
9 % Returns:
10 % pooledFeatures - matrix of pooled features in the form
11 %                  pooledFeatures(poolRow, poolCol, featureNum, imageNum)
12 %
13
14 numImages = size(convolvedFeatures, 4);
15 numFilters = size(convolvedFeatures, 3);
16 convolvedDim = size(convolvedFeatures, 1);
17 pooledFeatures = zeros(convolvedDim / poolDim, ...
18     convolvedDim / poolDim, numFilters, numImages);
19 % Instructions:
20 % Now pool the convolved features in regions of poolDim x poolDim,
21 % to obtain the
22 % (convolvedDim/poolDim) x (convolvedDim/poolDim) x numFeatures x numImages
23 % matrix pooledFeatures, such that
24 % pooledFeatures(poolRow, poolCol, featureNum, imageNum) is the
25 % value of the featureNum feature for the imageNum image pooled over the
26 % corresponding (poolRow, poolCol) pooling region.
27 %
28 % Use mean pooling here.
29
30
31 for i = 1:numImages
32     for j = 1:numFilters
33         for z = 1:(convolvedDim/poolDim)
34             for y = 1:(convolvedDim/poolDim)
35                 %Keeping the step 1 but configuring the range like current
36                 %place in the map multiplied by poolDim. We subtract the
37                 %poolDim from the starting position but not from the ending
38                 %position so we are configuring the range like this. Also
39                 %in the starting position we are adding 1 since in matlab
40                 %the counting starts from 1. We use the same formula in
41                 %both dimensions.

```

```

42         temp_matrix = convolvedFeatures((z*poolDim)-poolDim+1:z*poolDim, ...
43             (y*poolDim)-poolDim+1:y*poolDim,j,i);
44         pooledFeatures(z,y,j,i) = mean2(temp_matrix);
45     end
46 end
47 end
48 end
49
50 %alternative way
51 %%% Add code here
52 %     for i=1:numImages
53 %         for j=1:numFilters
54 %             x=1;
55 %             y=1;
56 %             for k=1:poolDim:convolvedDim
57 %                 for l=1:poolDim:convolvedDim
58 %                     pooledFeatures(x,y,j,i) = mean2(convolvedFeatures...
59 %                         (k:k+poolDim-1,l:l+poolDim-1,j,i));
60 %                     if(y==(convolvedDim / poolDim))
61 %                         x=x+1;
62 %                         y=1;
63 %                     else
64 %                         y = y+1;
65 %                     end
66 %                 end
67 %             end
68 %         end
69 %     end
70 end

```

Listing 2: cnnPool.m

6 Forward Pass

```

1 function [cost, grad, preds, activations] = cnnCost(theta,images,labels,numClasses,...
2             filterDim,numFilters,poolDim,pred)
3 % Calcualte cost and gradient for a single layer convolutional
4 % neural network followed by a softmax layer with cross entropy
5 % objective.
6 %
7 % Parameters:
8 % theta      - unrolled parameter vector
9 % images     - stores images in imageDim x imageDim x numImges
10 %             array
11 % numClasses - number of classes to predict
12 % filterDim  - dimension of convolutional filter
13 % numFilters - number of convolutional filters
14 % poolDim    - dimension of pooling area
15 % pred       - boolean only forward propagate and return
16 %             predictions
17 %
18 %
19 % Returns:
20 % cost       - cross entropy cost

```

```

21 % grad      - gradient with respect to theta (if pred==False)
22 % preds     - list of predictions for each example (if pred==True)
23
24
25 if ~exist('pred','var')
26     pred = false;
27 end;
28
29
30 imageDim = size(images,1); % height/width of image
31 numImages = size(images,3); % number of images
32
33 %% Reshape parameters and setup gradient matrices
34
35 % Wc is filterDim x filterDim x numFilters parameter matrix
36 % bc is the corresponding bias
37
38 % Wd is numClasses x hiddenSize parameter matrix where hiddenSize
39 % is the number of output units from the convolutional layer
40 % bd is corresponding bias
41 [Wc, Wd, bc, bd] = cnnParamsToStack(theta,imageDim,filterDim,numFilters,...
42                                     poolDim,numClasses);
43
44 % Same sizes as Wc,Wd,bc,bd. Used to hold gradient w.r.t above params.
45 Wc_grad = zeros(size(Wc));
46 Wd_grad = zeros(size(Wd));
47 bc_grad = zeros(size(bc));
48 bd_grad = zeros(size(bd));
49
50 %=====
51 %% Forward Propagation
52 % In this step you will forward propagate the input through the
53 % convolutional and subsampling (mean pooling) layers. You will then use
54 % the responses from the convolution and pooling layer as the input to a
55 % standard softmax layer.
56
57 %% Convolutional Layer
58 % For each image and each filter, convolve the image with the filter, add
59 % the bias and apply the sigmoid nonlinearity. Then subsample the
60 % convolved activations with mean pooling. Store the results of the
61 % convolution in activations and the results of the pooling in
62 % activationsPooled. You will need to save the convolved activations for
63 % backpropagation.
64 convDim = imageDim-filterDim+1; % dimension of convolved output
65 outputDim = (convDim)/poolDim; % dimension of subsampled output
66
67 % convDim x convDim x numFilters x numImages tensor for storing activations
68
69 %%% REPLACE THE FOLLOWNG LINE %%%
70 activations = cnnConvolve(filterDim, numFilters, images, Wc, bc);
71
72 % outputDim x outputDim x numFilters x numImages tensor for storing
73 % subsampled activations

```

```

74
75 %%% REPLACE THE FOLLOWNG LINE %%%
76 activationsPooled = cnnPool(poolDim,activations);
77
78 % Reshape activations into 2-d matrix, hiddenSize x numImages,
79 % for Softmax layer
80 %%% REPLACE THE FOLLOWING LINE %%%
81 activationsPooled = reshape(activationsPooled,[],numImages);
82 %alternative way
83 %activationsPooled = reshape(activationsPooled, outputDim*outputDim*numFilters,
    numImages);
84 %%% Softmax Layer
85 % Forward propagate the pooled activations calculated above into a
86 % standard softmax layer. For your convenience we have reshaped
87 % activationPooled into a hiddenSize x numImages matrix. Store the
88 % results in probs.
89
90 % numClasses x numImages for storing probability that each image belongs to
91 % each class.
92
93 %%% COMPUTE THE SOFTMAX OUTPUT %%%
94 probs = zeros(numClasses,numImages);
95 Ywx = Wd*activationsPooled;
96 Ywxb = bsxfun(@plus,Ywx,bd);
97 Ynum = exp(Ywxb);
98 probs = bsxfun(@rdivide,Ynum,sum(Ynum,1));
99
100 %=====
101 %%% STEP 1b: Calculate Cost
102 % In this step you will use the labels given as input and the probs
103 % calculate above to evaluate the cross entropy objective. Store your
104 % results in cost.
105 indexes = sub2ind(size(probs), labels', 1:numImages);
106 cost = -mean(log(probs(indexes)));
107
108 % Makes predictions given probs and returns without backproagating errors.
109 [~,preds] = max(probs,[],1);
110 preds = preds';
111 if pred
112     grad = 0;
113     return;
114 end;
115
116 %=====
117 %%% STEP 1c: Backpropagation
118 % Backpropagate errors through the softmax and convolutional/subsampling
119 % layers. Store the errors for the next step to calculate the gradient.
120 % Backpropagating the error w.r.t the softmax layer is as usual. To
121 % backpropagate through the pooling layer, you will need to upsample the
122 % error with respect to the pooling layer for each filter and each image.
123 % Use the kron function and a matrix of ones to do this upsampling
124 % quickly.
125

```

```

126 deriv = probs;
127 deriv(indexes) = deriv(indexes) - 1;
128 deriv = deriv ./ numImages;
129
130 Wd_grad = deriv * activationsPooled';
131 bd_grad = sum(deriv, 2);
132
133 deriv2_pooled = Wd' * deriv;
134 deriv2_pooled = reshape(deriv2_pooled, outputDim, outputDim, numFilters, numImages);
135 delta_upsampled = zeros(convDim, convDim, numFilters, numImages);
136
137 for im_idx=1:numImages
138     im = squeeze(images(:,:,im_idx));
139     for f_idx=1:numFilters
140         delta_pool = (1/poolDim^2) * kron(squeeze(deriv2_pooled(:,:,f_idx,im_idx)), ones
            (poolDim));
141         delta_upsampled(:,:,f_idx, im_idx) = delta_pool .* ...
            activations(:,:,f_idx,im_idx).*(1-activations(:,:,f_idx,im_idx));
142         delta_pool_sqz = squeeze(delta_upsampled(:,:,f_idx,im_idx));
143         cur_grad = conv2(im, rot90(delta_pool_sqz, 2), 'valid');
144
145         Wc_grad(:,:,f_idx) = Wc_grad(:,:,f_idx) + cur_grad;
146         bc_grad(f_idx) = bc_grad(f_idx) + sum(delta_pool_sqz(:));
147     end
148 end
149
150
151 %% Unroll gradient into grad vector for minFunc
152 grad = [Wc_grad(:) ; Wd_grad(:) ; bc_grad(:) ; bd_grad(:)];
153
154 end

```

Listing 3: cnnCost.m

7 Experiments

1. Our network starts with the imageDim parameter that its the input image dimension(W), creating a matrix imageDim x imageDim. The parameter numClasses is the output of the network, that basically is a vector numClasses x 1, that contains the possibilities that our input is associated with one of these classes(e.g. 10 digits 1 class for each, the sum of the possibilities must be equal to 1, the biggest possibility is the class that our input is associated). The parameter filterDim is the filter dimension. More specifically, this is responsible for creating a square filter with dimensions filterDim x filterDim x imageDepth (imageDepth = 3 for RGB, = 1 for Grayscale, every neuron is connected with a part-filter on the image, using the full image Depth), that we are using for iteratings over the different spatial region of the image. The numFilters parameters is the number of different features we are looking for in each different spatial region of the image, using our filters. The parameter poolDim is responsible for downgrading the output volume, e.g. for poolDim 2 in an output volume 28 x 28 x numFilters, we will have a 14 x 14 x numFilters result as new output volume. In our example, we are using batch combined with stochastic gradient training, as we are not using all the training set at once(batch), but neither every single training set example alone. Instead we are using small batches of images for training. Our cnnConvolve function is responsibe for the output volume, so every output of this function is the activation-output volume for a specific batch of images, all the filters applied in each and its structure is outputVolumeDim(imageDim - filterDim + 1) x outputVolumeDim x numFilters x numIm-

ages(Images for the current batch). After these, we are passing this matrix as a parameter to our pooling layer, in order to downgrade the image. Then we are reshaping, the first three dimensions into a continuous vector for each image, so now we have a two dimensional matrix, that contains the data for each image. Last step of forward passing is computing the propabilities for each image(which class is associated with each image more). Softmax layer is responsible for this, as it is using the current weight matrix, the biases and the input vector for each image to calculate the result. After this we are calculating the cost, and we proceed to the back propagation, in order to continue our network training. Using the built-in min-FuncSGD matlab function, we are passing as parameters our cost function(cnnCost.m) with the rest of the network parameters, and we are using as option 3 epochs with batches of 256 images and a learning rate of 1e-1. After these steps, we are testing our accuracy(answer at 7.3 about the results). The procedure of learning and testing is finished.

2. In the beginning of the training we see that our filters look like random noise patches but as our training progresses we see that the filters achieve more structure and finally after 3 epochs we can see clearly on them shapes e.g.(lines, angles, curves) that they can recognize on images that we feed to our network.

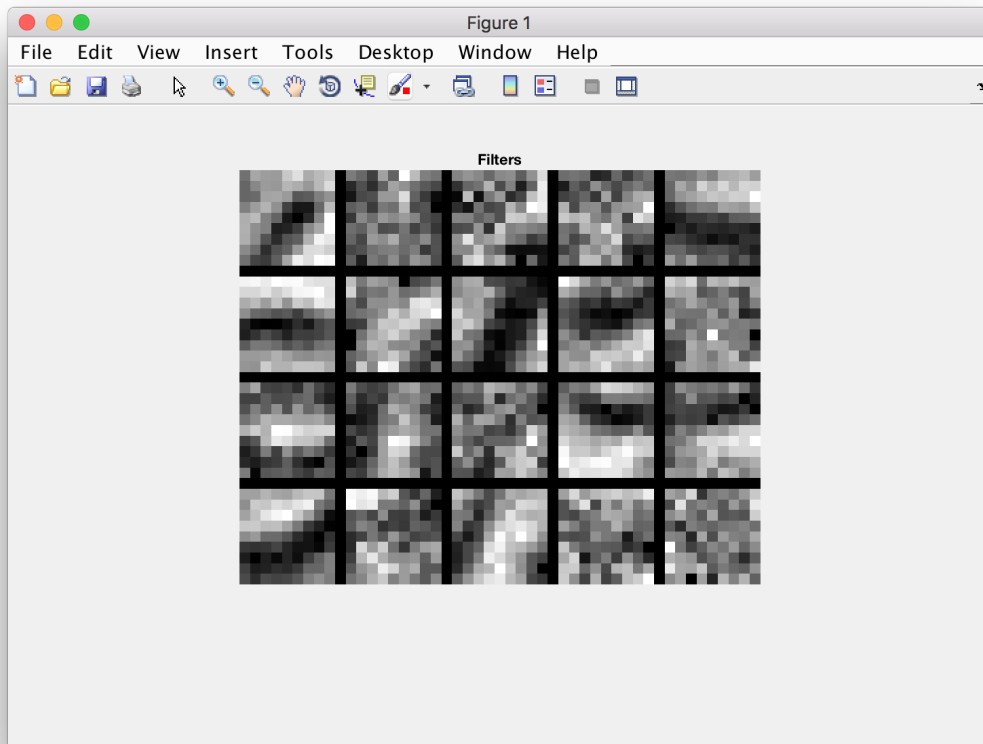


Figure 1: Filters after 3 epochs

3. Our network accuracy after 3 epochs is 97.24%