Neural Networks

Lab 4: Convolutional Neural Networks

1 Introduction

In this lab we will have a look at convolutional neural networks (CNNs). The idea of the CNN in its current form was initially proposed in 1998 by Yann LeCun. At that time, computational power and optimization techniques did not meet the requirements of modern CNNs. Currently, CNNs are trained using GPU acceleration and sophisticated optimization and regularization procedures. Unfortunately, we will not be able to use GPU acceleration for our networks, which is why we are going to implement a CNN with just one convolutional layer.

Before you start working on your own convolutional neural network, you should have read the following theory on CNNs: http://cs231n.github.io/convolutional-networks/. Study at least the sections until the 'Normalization layer'. The text discusses CNNs in context of the CIFAR-10 images database.

2 Theory questions (2 pt.)

1. Name two purposes of pooling and explain them using your own words.
2. Explain the importance of weight sharing in your own words.
3. Explain why using a rectified linear unit (ReLU) activation function might yield better results in less time than a sigmoid or tanh activation function.
4. Suppose we are given an image of 12×12 pixels with gray-scale values.
   (a) Let's assume we have a convolutional layer connected to the input image with 3 filters with dimensions 3×3. No zero-padding is used and the stride is 1. How many neurons does this layer contain?
   (b) Multi-layer perceptrons contain hidden layers that are fully connected to their respective preceding layer. In other words, every neuron in the hidden layer is connected to *all* neurons of the preceding layer.

   Suppose we have a hidden layer with the same amount of neurons as found in (a). How many connections are there between the input layer and the hidden layer? Compare this with the number of parameters for the layer in (a). What can you conclude?
5. Suppose we are given a dataset about the acceptance of a car. It can be labeled unacceptable, acceptable, good or very good. The features are price, maintenance costs, the number of doors, the capacity in number of persons, the lug boot size and a safety measure graded between 0 and 5.

   Since we have 6 features, we might want to arrange the features in 2×3 images. We could now implement a CNN to classify the acceptance. Explain why this approach is questionable.

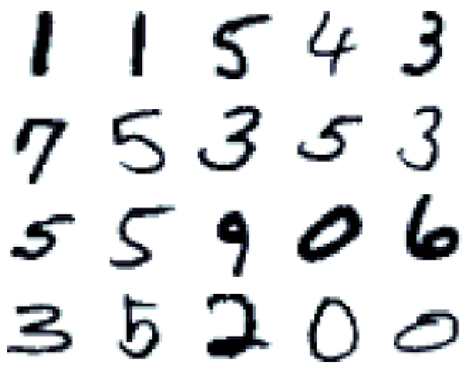
Figure 1: Sample from the MNIST dataset.

3 Classifying handwritten digits

Our CNN will learn to classify handwritten digits. Figure 1 displays a sample from the MNIST dataset. This dataset has been widely used as a benchmark for image processing systems, particularly machine learning applications. The database consists of 60.000 train images and 10.000 test images. The image dimensions are 28×28. All images are labeled, which makes the MNIST database suitable for supervised learning algorithms. By looking at Figure 1 it becomes clear that classifying handwritten digits is a nontrivial task. The MNIST dataset is included in the zip file that is provided on Nestor.

3.1 CNN architecture

The CNN that you are going to use will have (i) a convolutional layer, (ii) a mean pooling layer and (iii) a softmax output. An overview of the network is given in Table 1. The number of neurons in the input layer is simply $28 \times 28 \times 1$. For the convolutional layer we have dimensions $20 \times 20 \times 20$ and so on.

Table 1: Overview of the convolutional neural network.

Layer	Rows	Columns	# filters	batch size
Input (image)	28	28	NA	256
Convolution	20	20	20	256
Mean pooling	10	10	20	256
Reshaped mean pooling	2000	NA	NA	256
Softmax out	10	NA	NA	256

The last column is marked gray, because – in a way – it is redundant information. However, I still included it, because it is very important that you understand the dimensions here. The fact that we have multiple images in our batch implies that we have to use another dimension for storing the activations per image. We will come back to the batch size of 256 shortly.

3.2 Softmax output

The network will have 10 output neurons. Each neuron ‘votes’ for a certain digit. The output function is a so-called softmax function, which basically gives the posterior probabilities for the digits. In simpler terms, it will provide a measure of how confident it is about observing each digit, such that the sum of confidences is 1. The output of the ith output neuron is given by:

$$\hat{y}_i = \frac{\exp\left(\boldsymbol{w}_i \cdot \boldsymbol{x} + b_i\right)}{\sum_j \exp\left(\boldsymbol{w}_j \cdot \boldsymbol{x} + b_j\right)} \tag{1}$$

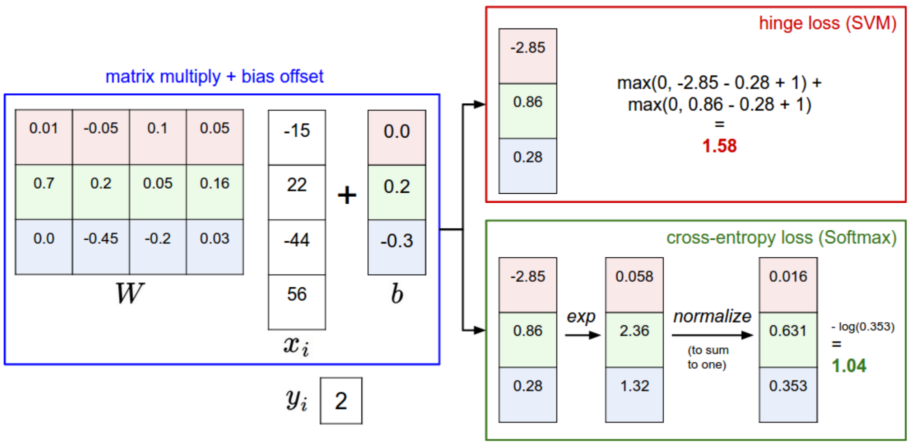

Figure 2: Illustration of softmax layer. You can ignore the part about Hinge loss.

where $\boldsymbol{w}_i$ is the ith weight vector between the mean pooling layer and the output layer. The bias b_i is similar to the threshold θ that we have seen in the MLP and the TLU, but its sign is changed. Note that this should also change the sign in its gradient, so the optimization problem is equivalent.

The output of the mean pooling layer is reshaped into a column vector $\boldsymbol{x}$. Figure 2 might be a helpful illustration of a matrix multiplication followed by a softmax layer. The final vector on the right depicts the output vector. We can see that the network favors the middle neuron (in green).

3.3 Training procedure

We will train the weights using backpropagation and stochastic gradient descent with mini-batches. In batch learning, you would normally train over all data points at once. Batch learning is guaranteed to converge to some minimum, while it requires more time to compute one update, so it can be less efficient than stochastic gradient descent. By using mini-batches, we effectively make a compromise between the efficiency of batch learning and the accuracy of stochastic gradient descent. We consider mini-batches of 256 images at a time. The training procedure also uses momentum, weight decay and a learning rate schedule. Inspect the code to see how these parameters are configured.

The back-propagation and training configurations are already implemented. You will have to implement the convolutional layer and the mean pooling layer yourself.

4 Implementing the convolutional layer (2 pt.)

Download the zip file on Nestor containing the starter code and the MNIST dataset. We are going to implement `cnnConvolve.m`. Read the top part of the function to see what we are going to do:

```
function convolvedFeatures = cnnConvolve(filterDim, numFilters, images, W, b)
%cnnConvolve Returns the convolution of the features given by W and b with
%the given images
%
% Parameters:
%  filterDim - filter (feature) dimension
%  numFilters - number of feature maps
%  images - large images to convolve with, matrix in the form
%           images(r, c, image number)
%  W, b - W is of shape (filterDim,filterDim,numFilters)
%         b is of shape (numFilters,1)
%
% Returns:
%  convolvedFeatures - matrix of convolved features in the form
%                      convolvedFeatures(imageRow, imageCol, featureNum, imageNum)

numImages   = size(images, 3);
imageDim    = size(images, 1);
convDim     = imageDim - filterDim + 1;

convolvedFeatures = zeros(convDim, convDim, numFilters, numImages);
```

4.1 Four dimensions

The function's output is a 4-dimensional array. The first and second dimensions are the rows and columns of the images respectively. The third dimension is for the filter index and the fourth dimension is for the image index of our mini-batch.

Here is a toy example to merely demonstrate the four dimensions. After we are done we could select a single filter output, that is, all rows and all columns from the 2nd filter output of the 73rd image as follows:

`single_filter_output = convolvedFeatures(:, :, 2, 73)`

The colons ':' allow us to select *all* rows and all columns of the 2nd filter output of the 73rd image in the mini-batch.

4.2 Formal definition

The formal definition of the forward pass of a convolutional layer as given in the slides is:

The output feature is given by multiple convolutions summed over the full depth of the input using the flipped weights plus some bias passed to an activation function

$$Y_j = f\left(b_j + \sum_{i=1}^{D} \mathring{W}_i^{(j)} * X_i\right), \; j = 1, 2, \ldots K \tag{2}$$

In this case $D = 1$, because the image only has gray-scale values. The j-index is used for the filter. In our network we will use the sigmoid activation:

$$f(x) = \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{3}$$

4.3 The algorithm

Recall that we have 20 filters and 256 images. The `cnnConvolve` should then perform 20×256 convolutions in each iteration. We will use Matlab's built-in `conv2` function that is well-optimized for doing this. Proceed as follows:

1. Create a `for` loop from 1 through `numImages`.
2. Store the current image from `images` in a matrix `im`.
3. Create a `for` loop from 1 through `numFilters`.
4. Now compute the convolved feature as given in equation (2) by taking the current filter from `W` and computing the convolution of this filter with `im`. Use MATLAB's built-in convolution function `conv2`.

Do not forget to flip the weights first! You can use MATLAB's `rot90` to flip the matrix horizontally and vertically. The lecture slides explain why this is necessary.

Note that we want to compute a 'valid' convolution. Check out the documentation of `conv2` to see how to do this.

5. Store the result in the correct location of `convolvedFeatures`.

You can test whether your convolution works as expected by calling `cnnExercise`.

5 Implementing the mean pooling layer (2 pt.)

We will now implement the mean pooling layer. We will have to complete the code in `cnnPool.m`. Read the top part of the function to see what the parameters are, and what they are for:

```
function pooledFeatures = cnnPool(poolDim, convolvedFeatures)
%cnnPool Pools the given convolved features
%
% Parameters:
%  poolDim - dimension of pooling region
%  convolvedFeatures - convolved features to pool (as given by cnnConvolve)
%                      convolvedFeatures(imageRow, imageCol, featureNum, imageNum)
%
% Returns:
%  pooledFeatures - matrix of pooled features in the form
%                   pooledFeatures(poolRow, poolCol, featureNum, imageNum)
%

numImages       = size(convolvedFeatures, 4);
numFilters      = size(convolvedFeatures, 3);
convolvedDim    = size(convolvedFeatures, 1);

pooledFeatures  = zeros(convolvedDim / poolDim, ...
        convolvedDim / poolDim, numFilters, numImages);
```

We will implement non-overlapping mean pooling. If for example we have a 4×4 image and we have a `poolDim` of 2, we will have four 2×2 patches. Hence, the output will be a 2×2 image.

In this case we want to perform convolution for all images in our batch and all filter outputs for each image. You should proceed as follows:

1. Create a loop for going through all images in the mini-batch.
2. Create a loop for going through all filters.
3. Pool the features of each filter and store the result in `pooledFeatures`. Note that you might need more loops to do so. You can use MATLAB's `mean2` function to compute the mean of the elements of a matrix.

When you think you're done, you can check whether you've implemented it correctly by running `cnnExercise.m`.

6 Implementing the forward pass (2 pt.)

Now let's put things together. At each iteration the network will compute the cost of its predictions on the current batch. During this computation it needs to perform a forward- and backward pass. At this point, the forward pass is not implemented yet. The forward propagation should be computed in the file `cnnCost.m`.

1. Read the file until the initialization of `activations`. Rewrite this line such that `activations` obtains the convolved features.
2. Now pool the features and store the result in `activationsPooled`.
3. The next layer requires the input for one image to be a vector. This means we have to reshape our 4D `activationsPooled` of size $10 \times 10 \times 20 \times 256$ to a 2D matrix of size 2000×256. Use MATLAB's `reshape` to do this.

The next steps will make clear that MATLAB is a programming language designed with mathematics in mind. You should not reside to using for-loops whatsoever: everything can be done in a few one-liners!

4. Compute the output of the network. Recall that there are 10 output neurons. The output of the ith neuron is given by the softmax function:

$$\hat{y}_i = \frac{\exp\left(\boldsymbol{w}_i \cdot \boldsymbol{x} + b_i\right)}{\sum_j \exp\left(\boldsymbol{w}_j \cdot \boldsymbol{x} + b_j\right)} \tag{4}$$

   The weights for the output layer are stored in `Wd`. They are stored row-wise with respect to the output neurons. The bias is given by `bd`. Note that `Wd` is a 10×2000 matrix: rows correspond to digits and columns to the number of neurons that connect to this layer.

   The `activationsPooled` contains a 2000×256 matrix. The rows correspond to the neurons and the columns correspond to the image in the batch.

   The softmax layer should give 10 activations for each of the 256 images. Hence, it makes sense to store everything in a 10×256 matrix where the rows correspond to the predicted digit, and the columns correspond to the image in the batch.

   It is difficult to implement the computation of the softmax in just one line of MATLAB code. We have to split things up.

   (a) First, let us compute part of the numerator of equation (4). We will compute $\boldsymbol{w}_i \cdot \boldsymbol{x}$ for all neurons and all images in the batch in one shot by matrix multiplication. Note that the result will be:

$$\hat{Y}_{wx} = \begin{pmatrix} \sum_k w_{1,k}x_{k,1} & \sum_k w_{1,k}x_{k,2} & \cdots & \sum_k w_{1,k}x_{k,256} \\ \sum_k w_{2,k}x_{k,1} & \sum_k w_{2,k}x_{k,2} & \cdots & \sum_k w_{2,k}x_{k,256} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_k w_{10,k}x_{k,1} & \sum_k w_{10,k}x_{k,2} & \cdots & \sum_k w_{10,k}x_{k,256} \end{pmatrix} \tag{5}$$

   Where $w_{i,j}$ is the i,j-th element of the matrix `Wd` and $x_{i,j}$ is the i,j-th element of the reshaped `pooledFeatures` matrix.

   (b) Now let us compute the matrix $\hat{Y}_{wx,b}$ in which the bias is also added. We will now add b_i to each i-th row.

   Remark

   You can add `bd` efficiently by using `bsxfun` with `@plus` as the first argument. Look up the documentation of `bsxfun` to find out how to do this.

   (c) Now apply `exp` to the full matrix. After this we have the matrix that contains the numerator of equation (4). We will denote this matrix $\hat{Y}_{num}$.

   (d) Recall that each column of $\hat{Y}_{num}$ corresponds to a single image. Hence, we want to normalize the activations of the neurons of a single column, such that the sum equals 1. To accomplish this we divide by the sum of the elements in a column.

   Use `sum(Y_num,1)` to sum along the columns. Now use `bsxfun` again to divide by the sums per column. This time you will have to call `bsxfun` using `@rdivide` as the first argument. Again, check out the documentation of `bsxfun` for further details.

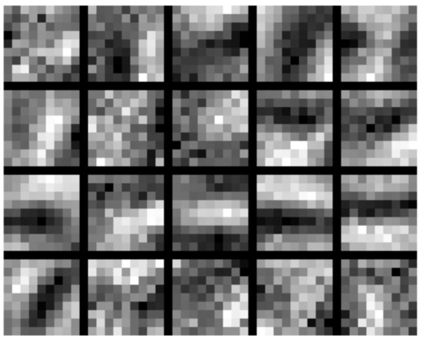

Figure 3: Illustration of 9×9 filters after 3 epochs of training.

(e) After dividing by the sums per column, we have obtained the softmax output for the full mini-batch. Store the result of $\hat{Y}$ in `probs`.

7 Experiments (1.5 pt.)

Once you are done with the above, we can begin training the network. You should be able to reach a test accuracy of $> 97\%$ after 3 epochs. The initial phase of the training procedure should give you output similar to this:

```
>> cnnTrain
Epoch 1: Cost on iteration 1 is 2.559325
Epoch 1: Cost on iteration 2 is 3.607505
Epoch 1: Cost on iteration 3 is 5.031685
Epoch 1: Cost on iteration 4 is 6.339704
Epoch 1: Cost on iteration 5 is 9.896953
Epoch 1: Cost on iteration 6 is 9.148324
Epoch 1: Cost on iteration 7 is 10.896994
Epoch 1: Cost on iteration 8 is 10.136041
Epoch 1: Cost on iteration 9 is 8.962045
Epoch 1: Cost on iteration 10 is 6.577781
Epoch 1: Cost on iteration 11 is 5.106075
Epoch 1: Cost on iteration 12 is 4.009335
Epoch 1: Cost on iteration 13 is 2.957203
Epoch 1: Cost on iteration 14 is 2.121366
Epoch 1: Cost on iteration 15 is 2.105306
Epoch 1: Cost on iteration 16 is 1.834664
Epoch 1: Cost on iteration 17 is 1.789108
Epoch 1: Cost on iteration 18 is 1.729627
Epoch 1: Cost on iteration 19 is 1.662368
Epoch 1: Cost on iteration 20 is 1.624098
```

There should also be an image of the 20 filters that are being trained. In the beginning, you will see that they more-or-less look like random noisy patches. After the first epoch you should see more structure in each filter.

1. Convince the reader of your report that you have understood the network. Mention all parameters that are relevant to the training procedure and what their role is. This includes

the parameters outside your own implemented pieces of code. *Do not explain formulas/loops etc.* Make sure your experiment is reproducible by anybody that at least understands CNNs.

2. Include the image of the filters after 3 epochs of training. See Figure 3 for an example. What can you say about the filters now when compared to the beginning of the training phase? Explain.

3. Report the accuracy of the network on the test set after 3 epochs. It should be higher than 97%.

8 What to hand in

Your report should include:

1. The answers to the theory questions.

2. The implementation of `cnnConvolve.m`, `cnnPool.m` and `cnnCost.m`.

3. The results of your experiments.