Natural computing

Assignment 4

Stijn Voss, s4150511
Kevin Jacobs, s4134621,
Jaap Buurman, s0828122

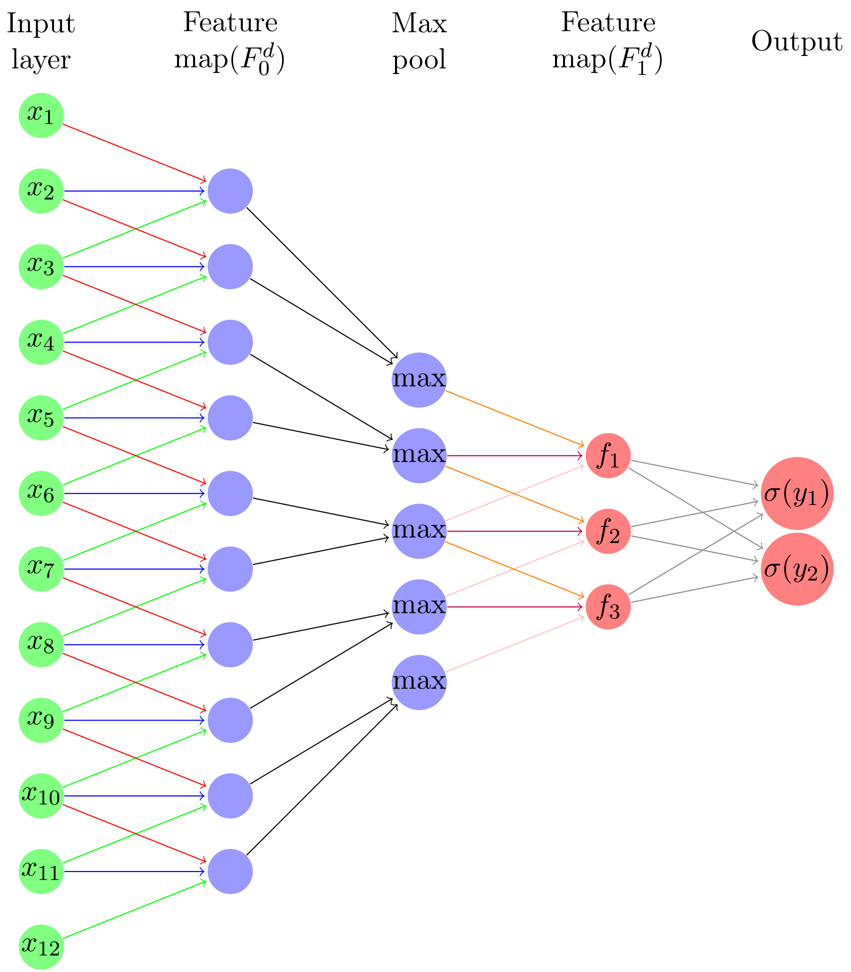
March 15, 2016

1 Exercise 1

Consider a convolutional neural network (CNN) with one-dimensional (1D) input. While two dimensional (2D) CNNs can be used for example for grayscale images, one-dimensional CNNs could be used for time-series such as temperature or humidity readings. Concepts for the 1D-case are equivalent to 2D networks. We interpret data in our network as three-dimensional arrays where a row denotes a feature map (also called filter, kernel), a column denotes a single dimension of the observation, and the depth of the array represents different observations. As we will only work with a single input vector, the depth will always be one. Let the following CNN be given:

- *Input I: Matrix of size 1 x 12 x 1. We therefore have an input with twelve dimensions consisting of a single feature map.*
- *First convolutional layer with filters* $F_0^1 = (1, 0, 1)$ *and* $F_1^1 = (1, 0, 1)$ *that generates two output feature maps from a single input feature map. Use valid mode for convolutions. Mode valid returns output of length max(M, N) - min(M, N) + 1, where M is the size of the array and N = 3 is the size of the filter. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.*
- *Max-pooling layer with stride 2 and filter size 2. Note that max-pooling pools each feature map separately.*
- *Convolutional layer with convolutional kernel* $F_0^2 = (1, 0, 1), (1, 0, 1)$ *of size 2 x 3 x 1.*
- *Fully connected layer that maps all inputs to two outputs. The first output is calculated as the negative sum of all its inputs, and the second layer is calculated as the positive sum of all its inputs.*

- *Sigmoidal activation function*
  - *(a) Draw the network.*
  - *(b) Calculate the response of the CNN for the two inputs* $(0;0;0;0;1;1;1;1;0;0;0;0)$ *and* $(1;1;1;1;0;0;0;0;1;1;1;1)$.



(a) The purple nodes are two-dimensional. If two non-black edges have the same color, same dimension of the from node and the same dimension for the to node than the weights of these edges are equal.

(b) The code used for calculating the network response is found in the appendix. First, the two convolutional filters (F_0^1 and F_1^1) are applied on the input layer.

In the tables below, the two feature maps can be found after applying the two convolution filters. The feature map on the left in convolution layer 1 is the feature map which is found after applying F_0^1 on the input layer. The feature map on the right in convolution layer 1 is the feature map which is found after applying F_1^1 on the input layer.

Layer	**First dimension**	**Second dimension**
Input layer	[0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0.]	
Conv. layer 1	[0. 0. 1. 1. 0. 0. -1. -1. 0. 0.]	[0. 0. -1. -1. 0. 0. 1. 1. 0. 0.]
Max-pooling layer	[0. 1. 0. -1. 0.]	[0. -1. 0. 1. 0.]
Conv. layer 2	[0. -4. 0.]	
Fully connected layer	4.0	-4.0
Output layer	0.9820137900379085	0.01798620996209156

Table 1: Network output for input $(0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0)$.

Layer	**First dimension**	**Second dimension**
Input layer	[1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1.]	
Conv. layer 1	[0. 0. -1. -1. 0. 0. 1. 1. 0. 0.]	[0. 0. 1. 1. 0. 0. -1. -1. 0. 0.]
Max-pooling layer	[0. -1. 0. 1. 0.]	[0. 1. 0. -1. 0.]
Conv. layer 2	[0. 4. 0.]	
Fully connected layer	-4.0	4.0
Output layer	0.01798620996209156	0.9820137900379085

Table 2: Network output for input $(1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1)$.

After that, max pooling with stride 2 and filter size 2 is applied on both the feature maps. Therefore, the resulting max-pooling layer also has 2 feature maps.

After applying max pooling, another convolutional layer is found. This time, it operates in 2D space and is applied on the concatenation of the two max-pooling outputs (so this makes a 2×5 matrix). The resulting output therefore is a 1×3 matrix.

Then, the positive and the negative sums are calculated in the fully connected layer. And on these sums, a sigmoid activation function is applied, which can be seen in the output layer.

2 Exercise 2

Convolutional Networks in practice. Consider the practical assignment by Andrea Vedaldi and Andrew Zisserman available at `http://www.robots.ox.ac.uk/ vgg/ practicals/cnn/`*. Perform the tasks and answer the questions highlighted in yellow in the text of the file tutorial cnn.pdf available in Black Board.*

Question. The third dimension of x is 3. Why?

The first two dimensions signify the number of pixels as height and width respectively. The final dimension of 3 signifies the number of channels per pixel. In this case, the channels signify the RGB values of each pixel.

Questions. Study carefully this expression and answer the following:

- Given that the input map x has $M \times N \times K$ dimensions and that each of the K filters has dimension $M_f \times N_f \times K$, what is the dimension of y?

  Since you are convolving the filters with the input, which requires the entire filter to overlap with the input, the size of the filter determines how big your output map y will be. The bigger the filter, the less number of steps you can take over the input while the filter is still completely overlapping with the input. The dimension of each output map y will be:

  $M - M_f + 1 \times N - N_f + 1 \times K - K + 1 = M - M_f + 1 \times N - N_f + 1 \times 1$

  As the third dimension of the input and the filter are equal, it is not possible to shift the filter in this dimension and thus this reduces to a singleton dimension. Since we are using $K^{'}$ number of filters, the total output will be:

  $M - M_f + 1 \times N - N_f + 1 \times 1 \times K^{'}$

  Removing the singleton dimension and plugging in the number from the exercise we get:

  $384 - 5 + 1 \times 512 - 5 + 1 \times 10 = 380 \times 508 \times 10$

- Note that x is indexed by $i + i^{'}$ and $j + j^{'}$, but that there is no plus sign between k and $k^{'}$. Why?

  Because as noted in the previous exercise we can slide the filter over the input in the M and N dimension since the filter is smaller than the input in these dimensions. The plus $i^{'}$ and plus $j^{'}$ denote the sliding of this windows over these dimensions. Since the third dimension of the input and filter are equal, sliding the filter in this dimension is not possible and thus we do not have a plus term here.

Question. Study the feature channels obtained. Most will likely contain a strong response in correspondences of edges in the input image x. Recall that w was obtained by drawing random numbers from a Gaussian distribution. Can you explain this phenomenon?

A Gaussian distribution is centered on zero. So if the filter is currently on top of a uniform surface with roughly the same brightness/color everywhere, the summing of the activation will roughly cancel out since the average of the filter's weight is roughly zero, as they were drawn from a Gaussian distribution, and they all get multiplied with roughly the same value. When a filter is on top of an edge, the color/brightness in that area will not be uniform and thus each weight gets multiplied by a different value. This causes the sum of these activation to have a high chance of being further away from zero and thus the final output is a more extreme value.

Questions.

- What filter have we implemented?

  We have implemented an edge detector. The weights of the filter sum up to be exactly zero. So if the filter is on top of an uniform surface with roughly the same color values everywhere in that area, the weights get multiplied by roughly the same values and thus they sum to zero. If the filter is on top of an edge, the weights get multiplied by different values and thus they no longer cancel.

- How are the RGB colour channels processed by this filter?

  The filter is just repeated three times in the third dimension. So the three color values are simply summed together. This means that it cannot discern different colors.

- What image structure are detected?

  As explained before, this filter is an edge detector. And thus for the example used, it will show the edges of the bell peppers.

Question. Some of the functions in a CNN *must* be non-linear. Why?

Because applying multiple linear transformation on the input can be simplified to a single linear transformation. This means that the only functions that the CNN can learn (they are function approximators), are linear functions. But the interesting functions that you would want to learn, such as a classifier, are highly nonlinear. And thus we need nonlinearities somewhere in order to be able to approximate nonlinear functions.

Questions.

- Run the code above and understand what the filter w is doing.

  The filter responds to the left most and right most pixels in opposite sign. So if these pixel values are equal they cancel out. If they are different due to a discontinuity in the color values because of an edge, the output is no longer zero. But since only the left and right pixels are compared in this filter, it can only detect vertical edges and not horizontal ones. The filter is duplicated in a second feature map, but with the signs switched. So on an edge, one feature map will output a negative value while the other one will output a positive value.

- Explain the final result of z.

  On z the rectified linear unit is applied. So while without this function one feature will output a negative value while the other one will output a positive value while the filter is on an edge, this transformation will zero out the negative response of the filter in one of the two channels. The final result is thus an output with 2 feature maps of vertical edges, where one feature map is active at the edge where a new object begins, while the other channel is active at the edge where the object ends.

Question. Look at the resulting image. Can you interpret the result?

The max pooling operation has a size of 15×15 in this example, and thus we will lose $15 - 1 = 14$ pixels in each dimensions, similarly to how we lose pixels when applying convolutional filters. The output at each location is the maximum of all three color channels over the entire pooling area of 15×15. This results in a down-sampled image.

Question. Understand what this operator is doing. How would you set κ, α and β to achieve simple L^2 normalization?

You would use a beta value of 1 so that the power in the denominator drops out. Kappa should be zero, so that the only term left is the sum of the x to the power of 2. Alpha can be set to any value, and denotes the amount of L^2 normalization.

Task.

- Open the file tinycnn.m and inspect the code. Convince yourself that the code computes the CNN just described.

  In the beginning of the file, the parameters are defined. It then runs a forward pass through the CNN which gives away the structure. x_1 is set to the input image x. It is then ran through a convolutional layer `(vl_nnconv)` to calculate x_2. The output of that operation is then ran through the pooling layer by calling `vl_nnpool` with x_2 to calculate x_3.

- Look at the paddings used in the code. If the input image x_1 has dimensions $M \times N$, what is the dimension of the output feature map x_3?

  We are using a 3×3 square filter, so without the padding the output of the convolutional layer will produce x_2 with dimensions $M - 3 + 1 \times N - 3 + 1 = M - 2 \times N - 2$.

But it is being padded by 1 pixel in both dimensions on both sides (so 2 pixels in total per dimension), so the actual output ends up with the exact same dimensions as the input, namely $M \times N$. Then x_2 is fed into the pooling layer. The size of the pooling filter is defined as 3×3 so again this will produce an output with dimensions $M - 3 + 1 \times N - 3 + 1 = M - 2 \times N - 2$. But here too there will be padding equal to the size of the pooling filter minus 1 divided by 2, which effectively counters the shrinking of the dimensions by the pooling by in this case padding the output by 1 pixel in both dimension on both sides. This again will result in an output x_3 of $M \times N$.

Task. Inspect *pos* and *neg* and convince yourself that

- *pos* contains a single true value in correspondence of each blob centre;

  It is a matrix with the same dimensions as the input image and it contains logical values. Visually inspecting where these are true shows a sparse pattern which matches with the dots in the left-most input picture.

- *neg* contains a true value for each pixel sufficiently far away from a blob.

  It is a matrix with the same dimensions as the input image and it contains logical values. Visually inspecting where these are true shows a lot of true values, which you would expect. The image is entirely black (false) everywhere, except for the spots where there is a dot in the left most input picture.

- Are there pixels for which both *pos* and *neg* evaluate to false?

  We can check this by running neg(neg == pos) == 0. This compares the two matrices to see where they have equal values. And then checks whether there are any false values at these locations. This shows that there are indeed a lot of pixels (79k) where both are set to false. This has to do with how *neg* is defined. *Neg* is not true at all locations where there is no blob, but rather it is only true in locations sufficiently far away from a blob. So in locations very close to a blob, but without an actual blob, *neg* will be false. But since there is no blob in that location, *pos* will be false as well.

Task. Train again the tiny CNN without smoothing the input image in pre-processing. Answer the following questions:

- Is the learned filter very different from the one learned before?

  Yes. The filter with normalization shows a very large negative weight in the middle, with slightly less but still negative weights to the right, left, up and down pixels relative to the middle. And it shows large positive weights for the four corners of the filter. When the smoothing is removed some of filter's corners or pixels adjacent to the middle are of opposite sign.

- If so, can you figure out what went wrong?

  The problem with not having smoothing is that there is no spatial smoothing equal size as the size of the convolutional filter. This smoothing allows the filter's activation to maximize exactly in the center of the blob instead of it being shifted sometimes. This shifting sometimes is caused by the center of a blob not perfectly matching with one pixel. Smoothing combats this.

- Look carefully at the output of the first layer, magnifying with the loupe tool. Is the maximal filter response attained in the middle of each blob?

  No. As argued in the question before, the maximum activation obtained is sometimes shifted by one pixel in a random direction from the center of the blob. This is caused by the center of the blob spanning multiple pixels.

Task. Train again the tiny CNN without subtracting the median value in preprocessing. Answer the following questions:

- Does the algorithm converge?

  No, it does not converge properly. The errors are still very large at the end of training and many pixels are misclassified.

- Reduce a hundred-fold the learning rate and increase the maximum number of iterations by an equal amount. Does it get better?

  It gets a lot better and produces results similar to the training with normalization. The big difference is that it took 100 times as long to train.

- Explain why adding a constant to the input image can have such a dramatic effect on the performance of the optimization.

  The neural network has to output a value of zero when there is no blob at that pixel. So the input values to the filter at each position without a blob multiplied by their weights should sum up to zero. With very large input values this is way more difficult to enforce, because larger values affect the filter's output much more. It needs a very precise filter to fulfill that condition. In order to reach that, a very small learning rate should be used so that the solution will not oscillate around the optimum. To combat the smaller learning rate, many more epochs are needed to successfully train the model.

Tasks.

- By inspecting initializeCharacterCNN.m get a sense of the architecture that will be trained. How many layers are there? How big are the filters?

The first layer is a convolutional layer with a filter size of 5×5. It contains 20 of such filters, resulting in 20 different feature maps. The stride is 1, which means the filter is moved pixel by pixel over the input.
The second layer is a maxpooling layer with a 2×2 size. The stride is 2, which means the filter gets moved by 2 pixels every step over the image. This effectively results in downsampling.
The third layer is again a convolutional layer, again with 5×5 filters, but this time applying the filters over all 20 feature maps from the previous layer. This results in $5 \times 5 \times 20$ filters. It has 50 of such filters and thus it outputs 50 different feature maps.
The fourth layer is again a maxpooling layer with the exact same specifications as the second layer.
The fifth layer is yet again a convolutional layer with $4 \times 4 \times 50$ filter size. It has 500 of such filters, and thus it outputs 500 feature maps.
The sixth layer is a ReLU layer, which adds a nonlinearity to the model.
The seventh layer is again a convolutional layer with filter size $2 \times 2 \times 500$. Is has 26 of such filters (one for each letter).
The eighth layer is the softmax layer. This results in an output of 26 values that sum up to one. These values can be interpreted as the probability that the neural net assigns to each letter.

- Use the function `vl_simplenn_display` to produce a table summarizing the architecture.

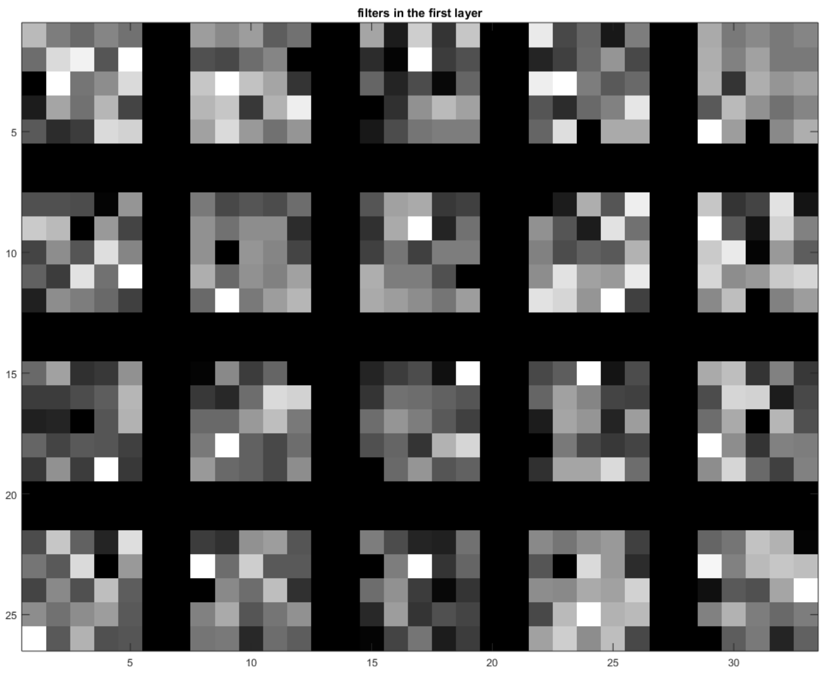
  See the next page for the table that it outputted.

Table 3: Specifications of the neural network

layer	1	2	3	4	5	6	7	8
type	cnv	mpool	cnv	mpool	cnv	relu	cnv	cnv
support	5x5	2x2	5x5	2x2	4x4	1x1	2x2	1x1
stride	1	2	1	2	1	1	1	1
pad	0	0	0	0	0	0	0	0
out dim	20	20	50	50	500	500	26	26
filt dim	1	n/a	20	n/a	50	n/a	500	n/a
rec. field	5	6	14	16	28	28	32	32
c/g net KB	2/0	0/0	98/0	0/0	1564/0	0/0	203/0	0/0
total								
network CPU/GPU memory: 1.8/0 MB								

Task. What can you say about the filters?

It is difficult to see, because the filters are rather noisy. But there seems to be some sort of edge detector structure in the filters.



Question. The image is much wider than 32 pixels. Why can you apply to it the CNN learned before for 32×32 patches?

Because convolutions can be applied regardless of the size of the input, as long as it is equal or smaller than the size of the filters that you are applying. The output of the filters will then be larger, but the output is divided into separate patches by sliding a window over it. Each patch separately is then classified, which outputs the probability of the patch containing each character out of 26 total characters. The model is trained on separate characters, but the model is applied to a sentence. For each 32×32 patch the model will make a prediction on what letter it is seeing.

3 Appendix

3.1 Code assignment 1

```
import numpy as np
from tabulate import tabulate
import math


def sigmoid(x):
    """
    Sigmoid!

    :param x: Input.
    :return:  sigmoid(x)
    """
    return 1.0 / (1.0 + math.exp(-x))


def apply_convolution(input, filter):
    """
    Apply 1D convolution.

    :param input:   The input vector.
    :param filter:  The convolution filter to apply.
    :return:        Convolution applied on the input vector (using mode valid)
    """
    m = input.shape[0]
    n = filter.shape[0]
    # Calculate the size of the vector after applying the convolution and usin
    size = max(m, n) - min(m, n) + 1
    output = np.zeros((1, size))
    # Calculate the convolution
    for i in range(0, len(input) - len(filter) + 1):
        s = 0
        for j in range(0, len(filter)):
            s += filter[j] * input[i + j]
        output[0, i] = s
    return output


def apply_max_pooling(input, stride, filter_size):
    """
    Apply max pooling.
    :param input:       An input vector.
    :param stride:      The stride.
    :param filter_size: The size of the max pooling filter.
    :return:            A vector produced by max pooling.
    """
    index = 0
    output = []
    while index + filter_size <= input.shape[1]:
        # Find the maximum of the items under the filter
        m = input[0, index]
        for filter_index in range(0, filter_size):
            m = max(m, input[0, index + filter_index])
        output.append(m)
        index += stride
    return np.array(output)


def show_network_output(x):
    """
    Show what the network computes for a given input.

    :param x: A 12 dimensional input vector.
    """

    # Apply convolution
    l_10 = apply_convolution(x, np.array([-1, 0, 1]))
    l_11 = apply_convolution(x, np.array([1, 0, -1]))

    # Apply max pooling
    l_20 = apply_max_pooling(l_10, 2, 2)
    l_21 = apply_max_pooling(l_11, 2, 2)

    # Apply 2D convolution using the 1D convolution method
    l_40 = apply_convolution(l_20, np.array([-1, 0, 1]))
    l_41 = apply_convolution(l_21, np.array([1, 0, -1]))
    l_4 = l_40 + l_41

    # Positive and negative sums
    y_1 = -l_4.sum()
    y_2 = l_4.sum()

    # Apply sigmoid
    o_1 = sigmoid(y_1)
    o_2 = sigmoid(y_2)

    # Convert it to a LaTeX table
    results = [['Input layer', x, '']]
    results.append(['Conv. layer 1', l_10[0], l_11[0]])
    results.append(['Max-pooling layer', l_20, l_21])
    results.append(['Conv. layer 2', l_4[0], ''])
    results.append(['Fully connected layer', y_1, y_2])
    results.append(['Output layer', o_1, o_2])
    print(tabulate(results, tablefmt='latex'))

# Calculate the network response of the two given inputs
show_network_output(np.array([0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0], dtype='floa
show_network_output(np.array([1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1], dtype='floa
```