Problem Set 2
Deep Learning E1394

Jorge Roa, 226454
Carlo Greß, 216319

Out on Oct 4, 2022
Due on Oct 18, 2022, at 23:59

Submit your written answers as a pdf typed in LaTeXtogether with your code. Submit one answer per group (as assigned on Moodle) and include names of all group members in the document. Round answers to two decimal places as needed. Include references to any external sources you have consulted (points are deducted if those were used but not cited). See "Submission" at the bottom of the problem set for more details on how to submit using Github classroom.

1 Convolutional neural networks

1.1 Kernels (7 pts)

(a) Design a 3×3 kernel that leaves the input image unchanged. Describe also how you may need to modify the input image before applying the kernel so that the output stays the same.

In order to leave the input image entirely unchanged, we need to use the identity kernel with the value 1 as the centre pixel and 0s in the other pixels:

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Regarding preprocessing/modification, we need to make sure that the output image has the same size (number of pixels) as the input. To achieve this goal, we can use zero padding. Here, we would add extra rows and columns of 0s to the input matrix. This makes sure that the output image has exactly the same amount of pixels (same row and column numbers) as the input. If we would not use same padding, the first entry in the upper left corner of the output would not be the same as the first entry in the upper left corner of the input (but rather the same as the entry in the second row/second column of the input).

(b) How does the following kernel modify an input image: $K = \frac{1}{16}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$?

We can easily see that the given kernel matrix has the same value between 0 and 1, and therefore, every pixel is an average of the surrounding pixels of the input. In praxis, the output picture look quite similar to the input, but it would appear blurry. Characteristic components like edges would still be visible, but not as pronounced as in the input. Since the filter is normalized (sum of all elements is 1), the overall intensity and brightness is preserved.

(c) Design a custom kernel of any size and describe how it transforms an input image or what it highlights in an image.

Opposite to the blurring kernel in (b), there also exist sharpening filters, which will emphasize extreme values of the input picture. Exemplarily, in a greyscale image, that would lead to more pronounced edges that emphasize color differences stronger than the original input. For a 3×3 kernel, that could look like this:

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

When calculating the output image, such a kernel would emphasize the center pixel while negatively weighting the surrounding pixels.

1.2 Convolution and pooling (15 pts)

1.2.1 Convolutional layer (10/15 pts)

Given an input image

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0.5 & 1 & 0.5 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \tag{1}$$

and a kernel $K = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 1 & 1 \\ 0 & 2 & 0 \end{bmatrix}$, compute the feature map (output after the convolution and applying a sigmoid activation function). Modify the input such that the feature map has the same dimension as the original input image. Show the intermediate result after the convolution and before applying the activation function as well.

For this task, we first apply padding in order to receive an output with the same size as the input. Hence, we are extending the given input matrix with surrounding 0s:

$$X_p = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0.5 & 1 & 0.5 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{2}$$

Now, we calculate the kernel $\tilde{K}$, where the entries are flipped to the opposite side. This step is not necessary but will make later calculations easier.

$$\tilde{K} = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

Now, we can calculate the convolution/cross correlation. After shifting the kernel over the input feature map and calculating the cross correlation, we receive the intermediate result:

$$O = \begin{bmatrix} 0 & 2 & 0 & 2 \\ 4 & 2 & 5 & 2 \\ 2 & 7.5 & 2.5 & 5.5 \\ 4 & 2 & 5 & 2 \end{bmatrix}, \tag{3}$$

Now, we are applying the sigmoid activation to the every value of the intermediate output matrix (output rounded to two decimal places):

$$O = \begin{bmatrix} 0.5 & 0.88 & 0.5 & 0.88 \\ 0.98 & 0.88 & 0.99 & 0.88 \\ 0.88 & 1 & 0.92 & 1 \\ 0.98 & 0.88 & 0.99 & 0.88 \end{bmatrix}, \tag{4}$$

1.2.2 Pooling layer (2/15 pts)

Apply 2×2 max pooling to the feature map (no overlap/stride 2).

When applying 2×2-max pooling to our 4×4-output, we will receive a final 2×2 result. This is due to the fact that we are not considering any overlapping columns/rows, but only consider the maximum values for each the upper/lower right/left corner (since the stride is 2). Hence, our final result is:

$$O = \begin{bmatrix} 0.98 & 0.99 \\ 1 & 1 \end{bmatrix}, \tag{5}$$

1.2.3 Discussion (3/15 pts)

Describe any problem(s) that you may see in training the model if a feature map like this one was typical for your CNN.

As we can see in the resulting matrix, every value is very close to the theoretical maximum of 1 after applying the sigmoid activation. Hence, comparing our final output matrix and our initial input, we can deduct that applying sigmoid and max pooling can lead to a loss of information, especially regarding pronounced differences between neighbouring pixels. This problem could be resolved by using a different pooling technique, for example average pooling. Secondly, the vanishing gradient problem could arise, since sigmoid's derivative tends to be very small and weights are only incrementally changing during the training process.

1.3 Pooling transformed into convolution (8 pts)

How do you represent 2×2 average pooling as a convolution? You may show this by the example of 4×4 input data. Provide the kernel size, kernel values, stride, etc. as appropriate.

In order to represent 2×2 average pooling as a convolution, the following steps are necessary:

Creating the kernel: Since we want to represent a 2×2 average pooling, the kernel should be of size 2×2. Since we are looking for the average, the kernel should have the following values:

$K = \frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

If our input picture is a 4×4 matrix, we can exemplarily use the matrix from section 1.2.1:

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0.5 & 1 & 0.5 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Now, we can apply the kernel to the input image. Since the output should be decreased in size (2×2-average pooling would lead to a 2×2 output map in case the input of a 4×4-input), we use stride equal to 2 for achieving a similar result. After computing the cross-correlation, we receive:

$$K = \begin{bmatrix} 0.25 & 0.25 \\ 2.5 & 2.5 \end{bmatrix}$$

1.4 Dimensions of CNN layers (10 pts)

For the CNN shown in Figure 1, write down the dimensions of each each layer and how you computed them. The input image is first increased to $3 \times 227 \times 227$ with padding.

Tip: A pooling layer with 'stride 2' means that after each pooling operation the next pooling area is 2 pixels apart. A 2×2 pooling operation with stride 2 would result in our example from class with no overlap. A 3×3 pooling operation with stride 2 has overlap. 'Pad 2' refers to padding with 2 pixels on each side.

`../img/PS2_network.png`

Figure 1: Convolutional neural network in simplified form. Note that the input image is first increased to $3 \times 227 \times 227$ with padding.

1. Layer 1: 11×11-convolution with 96 filters and stride 4.

   Here we, have 96 filters. The formula for the output after a convolution is calculated via

   $\frac{X-K+2Pa}{S} + 1,$

   where X is the input value, K is the kernel value, Pa is the padding value, and S is the stride. Inserting the given values of 96 filters, we receive:

   $\frac{227-11+0}{4} + 1 = 55$

Hence, the output size after the first layer is: $96 \times 55 \times 55$

2. Layer 2: 3×3 max pooling with stride 2.

   Plugging the values in the formula for the output after max pooling:

   $\frac{X-P_s}{S} + 1$, we receive:

   $\frac{55-3}{2} + 1 = 27$

   Hence, our output after the second layer is: $96 \times 27 \times 27$

3. Layer 3: 5×5 convolution with 256 filters and 2-padding.

   Plugging the values in the formula for convolutional layers again, we receive:

   $\frac{27-5+4}{1} + 1 = 27$

   Hence, our output after the third layer is: $256 \times 27 \times 27$

4. Layer 4: Max pooling with stride 2

   The fourth layer is identical to the second layer. We again use the formula for max pooling and plugging in the kernel and input values:

   $\frac{27-3}{2} + 1 = 13$

   Hence, our output after the fourth layer is $256 \times 13 \times 13$

5. Layer 5: 3×3-convolution with 384 filters and padding 1:

   Here, we are using the formula for convolutions again:

   $\frac{13-3+2}{1} + 1 = 13$

   Hence, our output after the fifth layer is $384 \times 13 \times 13$

6. Layer 6: 3×3-convolution with 384 filters and padding 1:

Layer 6 is the identical operation as in layer 5, and the result remains the same. Plugging in the values:

$\frac{13-3+2}{1}+1=13$

Hence, our output after the sixth layer is still $384 \times 13 \times 13$

7. Layer 7: 3×3-convolution with 256 filters and padding 1:

   After layer 7, only the filter size changes from 384 to 256. The rest of the output size is calculated via:

   $\frac{13-3+2}{1}+1=13$

   Our output after the seventh layer is $256 \times 13 \times 13$

8. Layer 8: 3×3-max pooling with stride 2:

   In layer 8, we apply a similar max pooling operation as in layers 2 and 4:

   $\frac{13-3}{2}+1=6$

   Our output after the eighth layer is $256 \times 6 \times 6$

   After these transformations, only fully connected layers are following. These already provide their respective size:

9. Layer 9: 1×4096

10. Layer 10: 1×4096

11. Layer 11: 1×1000

2 Monitoring power plant operations with CNNs

You will create an image classifier using PyTorch to estimate power plant operation. Specifically, you will learn how to apply CNNs to binary cassify LAWS' Sentinel 2 L2A satellite pictures (0: power plant is off, 1: power plant is on). All tasks are marked in their specific section and described in more detail in the Jupyter notebook. For some tasks, you may need to elaborate on your implementation. You can add the answers to these questions directly below the respective implementation task that you then upload to your GitHub repository. Remember to always show the code output in the final upload and use the "test your code" sections to test your implementation.

1. Implement your Convolutional Neural Network (20 pt)
2. Implement a hyperparameter tuner and fine-tune your network (20 pt)
3. Load and train a pre-trained model (10 pt)
4. Compare and discuss your results (10 pt)

Submission

The submission of the whole problem set is done via GitHub classroom. You have to register for the assignment with your GitHub account at `https://classroom.github.com/a/Ej3Bnsnb`. Please make sure that your GitHub account profile includes your real name. When starting the assignment, please create or join a team according to the teams assigned in Moodle (use the exact team name from Moodle). Please upload / push all solutions to the GitHub repository which was created for your team. Please upload your solutions for the theoretical part (1) as a PDF to GitHub. If you upload multiple PDF files, please indicate in the file name to which subtask they are corresponding (we much prefer one single pdf). For the practical part (2), just push your code changes to the existing files. Anybody in your team can push as often as they want. Once the deadline has passed, you are not able to push to your repository anymore and all changes on your **main** branch will be considered for grading. See the README.md in the repository for more details on how to push your changes to GitHub.