Assignment 6: Convolutional Neural Networks
MA-INF 2313: Deep Learning for Visual Recognition

Due Date: 24.01.2021

Assistants: Jan Müller, Leif Van Holland

Theoretical Task (15 pts)

a) Backpropagation through convolution & pooling *(8 points)*

Consider the following setting: Given is a quadratic input image $x \in \mathbb{R}^{n \times n}$ and a convolution kernel $w \in \mathbb{R}^{(2m+1)\times(2m+1)}$. Let $m \in \mathbb{N}$ and the indices of w be shifted such that $w_{0,0}$ is the value in the center of w. Then the discrete convolution $o = w * x$ of w and x is defined as

$$o_{i,j} = \sum_{k=-m}^{m} \sum_{l=-m}^{m} w_{k,l} \cdot x_{i-k,j-l} \text{ for } i,j = 1,...,n.$$

For out-of-bounds indices $i, j < 1$ or $i, j > n$, we assume that $x_{ij} = 0$, which coincides with padding the image with the appropriate number of zeros (also called `"SAME"` padding).

A max-pooling operation p on x with downsampling factor d can be defined as

$$p_d(x)_{i,j} = \max\left(x_{\pi_{ij}(1)}, ..., x_{\pi_{ij}(d^2)}\right)$$

where $\pi_{ij}(k)$ is a function that gives the appropriate indices from the input that are pooled. For example, given a 4×4 input image x and $d = 2$, we would have the following index mapping

$$\begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 \\ 2,1 & 2,2 & 2,3 & 2,4 \\ 3,1 & 3,2 & 3,3 & 3,4 \\ 4,1 & 4,2 & 4,3 & 4,4 \end{bmatrix} = \begin{bmatrix} \pi_{1,1}(1) & \pi_{1,1}(2) & \pi_{2,1}(1) & \pi_{2,1}(2) \\ \pi_{1,1}(3) & \pi_{1,1}(4) & \pi_{2,1}(3) & \pi_{2,1}(4) \\ \pi_{1,2}(1) & \pi_{1,2}(2) & \pi_{2,2}(1) & \pi_{2,2}(2) \\ \pi_{1,2}(3) & \pi_{1,2}(4) & \pi_{2,2}(3) & \pi_{2,2}(4) \end{bmatrix}$$

and a resulting pooled image

$$o' := p_2(x) = \begin{bmatrix} p_2(x)_{1,1} & p_2(x)_{1,2} \\ p_2(x)_{2,1} & p_2(x)_{2,2} \end{bmatrix}.$$

An MSE loss E in this two-dimensional setting is simply the mean of all n^2 squared differences. Because we are using `"SAME"` padding, the output has the same size as x, i.e. $o, y \in \mathbb{R}^{n \times n}$ and

$$E(o, y) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (o_{i,j} - y_{i,j})^2.$$

1. *(3 points)* Compute the partial derivative of $E(o, y)$ for the convolution $o = w * x$ with respect to the weights $w_{s,t}$ of the convolution kernel.

2. *(3 points)* Compute the partial derivative of $E(o', y')$ for the 2-max-pooling $o' = p_2(x)$ with respect to the inputs $x_{i,j}$. You may assume that n is even.

3. *(2 points)* Improve the definition of the convolution above by adding a single parameter γ for stride (i.e. the same stride in both image directions). Compute the partial derivative of $E(o, y)$ with respect to $w_{s,t}$ for the updated definition.

b) Receptive field and output sizes *(7 points)*
The receptive field of a filter in a CNN is the area in the input image that affects the outcome of the filter. For example, a filter of a 3×3 convolution layer applied directly on the input has a receptive field of 3×3. If we apply another 3×3 convolution on the output, the filters of the second layer will have a receptive field of 5×5. Similarly, for a pooling operation of factor 2 applied on the input, the receptive field of each pooled value will be 2×2.

1. *(2 points)* Come up with a formula that allows to compute the receptive field size after applying m convolution layers with square kernels of size $k_i \times k_i$ for $i = 1, ..., m$.

2. *(3 points)* As we will see in the programming part, usually CNNs are made out of blocks of multiple convolutions followed by a pooling operation. Expand your formula to compute the receptive field of m blocks of m_j convolution layers with square kernels of size $k_{ji} \times k_{ji}$ ($j = 1, ..., m$ and $i = 1, ..., m_j$, i.e. k_{ji} is the kernel size of the i-th convolution in the j-th block). The m pooling operations downscale their input by a factor of d_j.

3. *(2 points)* Give a similar formula that computes the output size of the net consisting of blocks like above for an input of size $n \times n$. Here you should consider the usual `"VALID"` padding, i.e. that the convolution is only applied at locations where the whole kernel fits inside the image. In other words, there is no (zero-)padding around the image.

Programming Task (15+2 pts)

a) A CNN for CIFAR-10 (*5 points*)
Finally, we are able to use CNNs to improve our CIFAR-10 classifier. In the following we simply use an Adam optimizer with a learning rate of $\eta = 0.0005$ and parameters $\beta_1 = 0.9, \beta_2 = 0.95$. Additionally, we will use a cross entropy loss function.

1. *(1 point)* As always, we should first of all establish a baseline. Recreate the model with batch normalization from last sheet, train the model for 25 epochs and record loss value and test accuracy after each epoch.

2. *(3 points)* Create a CNN consisting of two blocks: The first block should contain two convolution layers with a 3×3 kernel and 32 output channels each, followed by a 2×2 max-pooling. The second block should look the same, but should use 64 channels each. Then the output should be flattened and a linear layer should reduce the number of neurons to the number of classes. Add a ReLU activation after each layer. In total, your model should resemble this:

$$[conv_{32} \to conv_{32} \to maxpool] \to [conv_{64} \to conv_{64} \to maxpool] \to dense_{10}$$

Train the model for 25 epochs and record loss value and test accuracy after each epoch. *Hint:* If you struggle with finding the right input size for the linear layer, you can use your formula from above. You can also use torchsummary, which is a great tool to debug layer sizes of your model.

3. *(1 point)* Plot losses and accuracies of the two models above. Do you observe an improvement? If yes, why might the new CNN model be better suited for the classification setting?

b) Data augmentation (*5+2 points*)

Motivated by our success from a) we want to try to push the accuracy of our CNN even further. Unfortunately, the ampunt of training data we got is limited, but there is a way to get much more images for free.

1. *(1 point)* The torchvision.transforms package contains a multitude of ready-to-use random transformations to modify our training images. Check the list of available functions there and find 2-3 transformations that randomly produce new versions of images and fit our setting, i.e. they should produce outputs of the correct shape and that do not alter key features of the image (for example changing the color of the sky to green could already distort the image too much). State which transformations you chose and why.

2. *(1 point)* Refactor your dataset class in such a way that it applies your choice of transformations in the `__getitem__` method. Plot some test images to check if the transformations work the way you expect.

3. *(2 points)* Train the CNN using the new augmented dataset for 25 epochs and record loss and test accuracy after each epoch. You may have to experiment with the choice of parameters of the transformations to see an improvement.

4. *(1 point)* Plot losses and accuracies of the data-augmented model together with the results from above.

5. *(2 bonus points)* Try out if you can improve the test accuracy even further using any of the regularization methods we have discussed on previous sheets. Again, plot losses and accuracies as above, showing all results you obtained.

c) Visualizing filter responses (*5 points*)
A great way to debug the convolutions of the CNN is to take a look at the responses of the different channels.

1. *(1 point)* Get a random test image from the training dataloader and plot it.
2. *(2 points)* Run your trained CNN model from the previous exercise with this one image and obtain all intermediate values of the convolution layers (after the ReLU activation). An easy way to accomplish this is to use forward hooks. These are basically callback functions providing you with inputs and outputs of a `torch.nn.Module` when its `forward` function is called.
3. *(2 points)* Plot the activations for each layer in a 8×4 (for 32 channels) or 8×8 (64 channels) grid. Use the same scale for the color maps of each activation, so that the level of activation is comparable. What do you observe?