RWTH Aachen University
Institute for Data Science in Mechanical Engineering

Document Version 2.1.1, released February 11, 2022

Bonus Points Assignment II

Introduction

This assignment consists of three parts. In Parts 1 and 2, you will implement from scratch a convolutional neural network for image classification in NumPy. Part 1 implements the convolution operation, and Part 2 focuses on backpropagation. In Part 3, you will use PyTorch to train a neural network for image classification on an industrial dataset. Part 2 requires to complete Part 1, but Part 3 is independent from the other two. Part 2 is also harder than the other two.
You will find two Jupyter Notebooks accompanying this document; one for Parts 1 and 2, and one for Part 3.

Formalia You can earn up to 5% of bonus points for the final exam by completing this voluntary assignment. These bonus points will only be applied when passing the exam; a failing grade cannot be improved with bonus points.
This assignment starts on January 28th, and should be handed in before February 20th, 23:59:59 CET. You should solve the assignment in groups of 1 to 4 students. All group members have to register in the same group on Moodle under *Groups Assignment 2*.
Please hand in your solution as **a single zip file** including all notebooks and the `outputs` folder, which contains files automatically generated by your solutions. Before submitting, make sure that you have restarted the kernel of the notebook to clear all variables and run the notebook in one go. Do not clear the outputs.

Grading Grading may involve automatic unit tests. Ensure that your notebook runs correctly from start to finish, and that the output of your functions match the specifications.
We provide sample unit tests throughout the notebook to help you check your implementation. The tests used for grading will be different.

Python Setup To run the Jupyter Notebooks for this assignment, we recommend to use the RWTH JupyterHub. You are free to use other solutions if you wish to, such as a local installation, Google Colab, ... We will however not provide support for running the code on such alternatives.
Finally, Part 3 requires training a neural network on a dataset of intermediate size. The computing power available to you on the RWTH JupyterHub is sufficient to do so in a reasonable time; bear in mind that it might be slower on your own computer.

Required Libraries You will need to install `NumPy`, `Matplotlib`, and `PyTorch` to complete this assignment.

1. Forward Pass: Implementing Convolution

In this part, we will implement a convolution operation using `NumPy` only. The questions should be answered in the notebook `CSME2-WS21-BPA2-CNN_with_NumPy.ipnyb`.

Images and Kernels We will represent an image as an `np.ndarray` of shape `(B, h, w, F)`, where `h` and `w` are respectively the height and width of the image, and `F` is its number of filters. Recall that an RGB image has 3 filters, each corresponding respectively to red, green, and blue. Additionally, we will support batched inputs, and `B` is the batch size. As an example, the variable `images[b, :, :, 1]` is a 2D array containing all pixels of the second filter of image number $b \leq B-1$.
We will convolve such images with kernels. A kernel is an `np.ndarray` of shape `(K, K, F_in, F_out)`, where `K` is the kernel height and width, and `F_in` and `F_out` are respectively the number of input and output filters. For simplicity, we only consider square kernels.

Padding Images can be padded with 0 values on their side. We will assume that the same amount of padding is added on all sides and filters of the image; we denote by P the amount of padding on one side. Padding therefore increases the size of the image in each dimension by $2{\cdot}P$.

Convolution Assume we are given a padded input image of shape $(h_{\text{in}} + 2P, w_{\text{in}} + 2P, F_{\text{in}})$, denoted by $(x^{f_{\text{in}}}_{s,t})$. We transform it into an output image of shape $(h_{\text{out}}, w_{\text{out}}, F_{\text{out}})$, denoted by $(z^{f_{\text{in}}}_{p,q})$, with the convolution operation defined as follows:

$$z^{f_{\text{out}}}_{p,q} = \sum_{f_{\text{in}}=0}^{F_{\text{in}}-1} \sum_{i,j=0}^{K-1} k^{f_{\text{in}},f_{\text{out}}}_{i,j} \cdot x^{f_{\text{in}}}_{p\cdot S+i,\ q\cdot S+j}\ , \tag{1}$$

where S is the stride, and for all $(p, q, f_{\text{out}}) \in \{0, \dots, h_{\text{out}} - 1\} \times \{0, \dots, w_{\text{out}} - 1\} \times \{0, \dots, F_{\text{out}} - 1\}$. We reserve the indices i and j for indexing over the kernel, p and q for indexing over the output image, and s and t for indexing over the input image[1]. For this formula to be well defined, we need to have the following relation between the kernel size, input and output dimensions, and the padding and stride:

$$h_{\text{out}} = \frac{h_{\text{in}} + 2\cdot P - K}{S} + 1, \tag{2}$$

$$w_{\text{out}} = \frac{w_{\text{in}} + 2\cdot P - K}{S} + 1. \tag{3}$$

1.1. Convolutions and Image Filtering

We start by implementing the convolution operation, and demonstrate on an example how well-tuned kernels can be used for classical image filtering.

Question 1.1 Fill in the function `add_padding`.

[1] In convolution, we often have $s = p \cdot S + i$ and $t = q \cdot S + j$.

Identity	Sharpening	Edge Detection 1	Edge Detection 2	Edge Detection 3	Gaussian Blur
$\begin{pmatrix}0&0&0\\0&1&0\\0&0&0\end{pmatrix}$	$\begin{pmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{pmatrix}$	$\begin{pmatrix}1&0&-1\\0&0&0\\-1&0&1\end{pmatrix}$	$\begin{pmatrix}0&1&0\\1&-4&1\\0&1&0\end{pmatrix}$	$\begin{pmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{pmatrix}$	$\frac{1}{16}\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix}$

Table 1: The kernels to use in the function `filter_example`. They are well-known kernels from classical image processing (i.e., without neural networks), and are commonly used in standard image editing software.

Question 1.2 Fill in the function `convolve(kernels, images, S)`, which outputs the convolution between `kernels` and `images` as per Eq. (1). Assume that `kernels` and `images` have the correct dimensions for the convolution to be possible, and do NOT add padding in this function. Remember that:

- `images` is a batch of images;
- The output should be of shape $(B, h_{\text{out}}, w_{\text{out}}, F_{\text{out}})$.

Hint. *Keep things simple and use* `for` *loops. We discuss a more efficient implementation without such loops later, but this is significantly harder.*

Question 1.3 Fill in the function `filter_example`. This function applies each of the kernels defined in Table 1 on each of the color channels of the example image independently and plots the resulting RGB image. Use a padding and stride of 1.
More precisely, if `example_image` is the array of shape `(1, 100, 100, 3)` containing the example image, the output image should have the same shape, and its k-th filter should be the result of the convolution between the chosen kernel of Table 1 and the k-th filter of `example_image`. The plotting part is already implemented; your result should be as on Figure 1. In particular, make sure that your output does not have a noticeable color shift when visually compared with Figure 1.

Hint. *Use the* `convolve` *function defined earlier with a well-chosen kernel.*

1.2. Forward Pass

We will now use the functions `add_padding` and `convolve` to implement the forward pass of a convolutional neural network. We propose a modular architecture similar to the one of PyTorch. In our architecture, all operations will implement the abstract `Layer` interface. It defines (but does not implement) three methods:

- `forward(self, x)`: this is the forward pass of the layer. It receives the output of the previous layer (or the input to the network), applies its transformation, and returns the result;

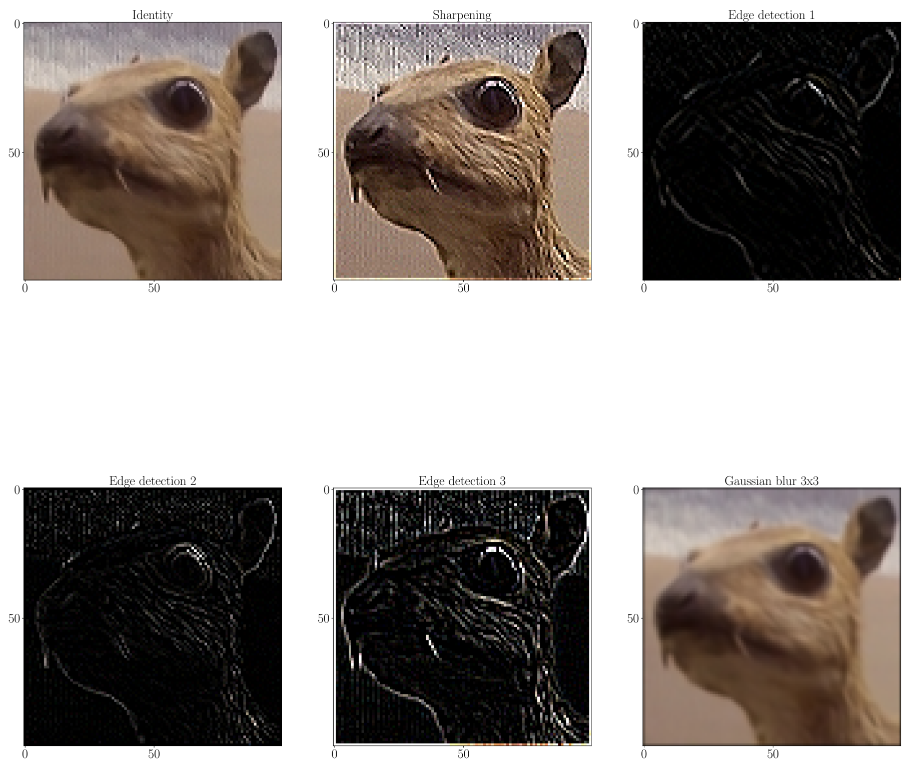


Figure 1: The result of the function `filter_example`. Notice how each kernel has a very specific action on the input image, such as sharpening, blurring, or edge detection. Source of the original image: [1]

- `backward(self, gradient)`: this is the backward pass of the layer. It receives the gradient of the next layer with respect to its input;
- `update(self, learning_rate)`: this method updates the weights of the layer according to the learning rate and the previously computed gradients. In particular, it should only be called after `backward`.

Remember our convention that `x` is of shape (B, h, w, F)[2], where B is the batch size, h the height, w the width, and F the number of filters. In particular, `x` should be of shape $(B, 1, 1, F)$ for fully connected layers.

Question 1.4 The `Conv2D` class implements a convolutional layer. It computes the convolution between its input `x` and kernel `self.kernels`, and adds a bias term independently for each output filter. In other words, the value of the output neuron (p, q) is given by:

$$z_{p,q}^{f_\text{out}} = k_\text{bias}^{f_\text{out}} + \sum_{f_\text{in}=0}^{F_\text{in}-1} \sum_{i,j=0}^{K-1} k_{i,j}^{f_\text{in},f_\text{out}} \cdot x_{p\cdot S+i,\; q\cdot S+j}^{f_\text{in}} \,, \tag{4}$$

Fill in the `forward` method of the `Conv2D` class.

Hint. *Don't forget the padding.*

[2]Typo fixed in v2.1.0

Question 1.5 Fill in the `forward` method of the `ReLU` class.

Question 1.6 Fill in the `forward` method of the `MaxPooling` class.

Hint. *You can't directly use the* `convolve` *function for this, but the implementation is extremely similar!*

Question 1.7 Fill in the `forward` method of the `FullyConnected` class.

Hint. *If* `A` *and* `B` *are* `np.ndarray`*s of shape* `(..., N, M)` *and* `(..., M, P)` *respectively, the array product* `A @ B` *(or equivalently,* `np.matmul(A, B)`*) is interpreted as the matrix product along the last two dimensions, and has shape* `(..., N, P)`*. Of course, the dimensions hidden in the* `...` *should match for this to be true, or be broadcastable. Refer to the NumPy documentation of* `matmul` *for more details.*

Question 1.8 Fill in the `forward` method of the `FeedForwardNet` class.

2. Implementing Backpropagation

In this section, we implement backpropagation for the network that we defined in Part 1. We will use our network for binary image classification, and therefore use the binary cross-entropy loss introduced in the lecture, which we denote by L. Additionally, we will use the notation L_n to denote the individual loss of a tranining example, such that:

$$L = \frac{1}{N} \sum_{n=1}^{N} L_n \tag{5}$$

For each layer, we propose to implement a `backward` method computing the gradients of the loss L_n with respect to:

- the input of the layer;
- the parameters of the layer.

In particular, layers that do not have parameters such as `MaxPooling` and `ReLU` only need to compute the first of these gradients. Other layers will store two parameters, `grad_w` and `grad_b`, each corresponding to the gradient w.r.t. the weights/kernel and the bias. They will also implement an `update` function, which computes the correct values of `grad_w` and `grad_b` and updates the weights and biases according to these gradients. The backward pass therefore consists in successively calling the `backward` method of each layer.
The questions in this section are independent, and in order of increasing difficulty.

Question 2.1 Fill in the `backward` and `update` methods of the `FullyConnected` class.

Question 2.2 Fill in the `backward` method of the `ReLU` class.

Hint. *The derivative is not defined at* 0*. See Lecture 12 for a discussion about an appropriate value.*

Question 2.3 Fill in the `backward` method of the `MaxPooling` class. You can assume that the kernel shape `K` and the stride `S` are equal, such that each input neuron is in exactly one pool. Under this assumption, the derivative of the output w.r.t. the input is given by:

$$\frac{\partial z_{p,q}}{\partial x_{s,t}} = \begin{cases} 1, & \text{if } \left(\frac{s-p}{S}, \frac{t-q}{S}\right) \in \{0, \ldots, K-1\}^2 \text{ and } x_{s,t} = z_{p,q} \\ 0, & \text{otherwise,} \end{cases}$$

where we omitted indexing over the filters since the operation acts independently over all filters. In other words, the output gradient is equal to the incoming gradient if the input neuron $x_{p,q}$ is selected in the maximum (i.e., if (s,t) is in the pool and $x_{s,t} = z_{p,q}$), and is equal to 0 otherwise.

Hint. *You may need to modify the* `forward` *method to keep track of what neuron achieved the maximum in each pool. If several neurons achieve the maximum, you can just keep the last one in the pool.*

In order to implement the `backward` method of the `Conv2D` class, we will need to dilate the incoming gradient to account for the stride of the forward pass. We start by implementing dilation of a generic image.

Question 2.4 Fill in the `dilate` function, which takes as input an image of shape $(B, h_{\text{in}}, w_{\text{in}}, F_{\text{in}})$ and a dilation size $D \in \mathbb{N}_>$, and outputs a dilated image of shape $(B, h_{\text{out}}, w_{\text{out}}, F_{\text{in}})$, where:

$$h_{\text{out}} = (h_{\text{in}} - 1) \cdot D + 1, \tag{6}$$

$$w_{\text{out}} = (w_{\text{in}} - 1) \cdot D + 1. \tag{7}$$

The dilation is defined mathematically as inserting $D-1$ neurons with 0 value between each neuron of the input image independently on each filter.

Hint. *Use the function* `np.repeat`*.*

We are now equipped to complete the `Conv2D` class. By taking derivatives in the equation of the convolutional layer (4), one can derive the expressions of the required gradients. These computations are involved; we provide the results below.

Gradient w.r.t. the Bias The gradient of the loss w.r.t. the bias term is given by:

$$\frac{\partial L_n}{\partial k_{\text{bias}}^{f_{\text{out}}}} = \sum_{p=0}^{h_{\text{out}}-1} \sum_{q=0}^{w_{\text{out}}-1} \frac{\partial L_n}{\partial z_{p,q}} \tag{8}$$

Gradient w.r.t. the Other Kernel Weights We only provide here the result of the computation of the gradient that enables implementation; we refer you to Appendix A if you are interested in the proof. Additionally, the website [2] provides a nice visual explanation for the proof.
Formally, we have:

$$\frac{\partial L_n}{\partial k_{i,j}^{f_{\text{in}},f_{\text{out}}}} = \sum_{s=0}^{h_{\text{in}}-1} \sum_{t=0}^{w_{\text{in}}-1} g_{s-i,t-j}^{f_{\text{out}}} \cdot x_{s,t}^{f_{\text{in}},f_{\text{out}}}, \tag{9}$$

where we have defined the dilation g:

$$g_{u,v}^{f_{\text{out}}} = \begin{cases} \frac{\partial L_n}{\partial z_{\frac{u}{S},\frac{v}{S}}}, & \text{if } (\frac{u}{S},\ \frac{v}{S}) \in \{0,\ldots,h_{\text{out}}-1\} \times \{0,\ldots,w_{\text{out}}-1\}, \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

In other words, this derivative is obtained by first dilating the incoming gradient $\frac{\partial L_n}{\partial z_{p,q}}$ with size S, and then computing the convolution of its f_{out}-th channel (used as kernel) with the f_{in}-th channel of the input.

Gradient w.r.t. the Input Here again, we only provide the result. The proof is very similar to the one in Appendix A, and we refer to the website [3] for a visual explanation.
Let us introduce the flipped kernel:

$$\kappa_{i,j}^{f_{\text{out}},f_{\text{in}}} = k_{K-1-i,K-1-j}^{f_{\text{in}},f_{\text{out}}}, \tag{11}$$

for all $i, j, f_{\text{in}}, f_{\text{out}}$. It is itself a kernel of shape $(K, K, F_{\text{out}}, F_{\text{in}})$. Formally, we have:

$$\frac{\partial L_n}{\partial x_{s,t}^{f_{\text{in}}}} = \sum_{f_{\text{out}}=0}^{F_{\text{out}}-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \kappa_{i,j}^{f_{\text{out}},f_{\text{in}}} \cdot \gamma_{s+i,t+j}^{f_{\text{out}}}, \tag{12}$$

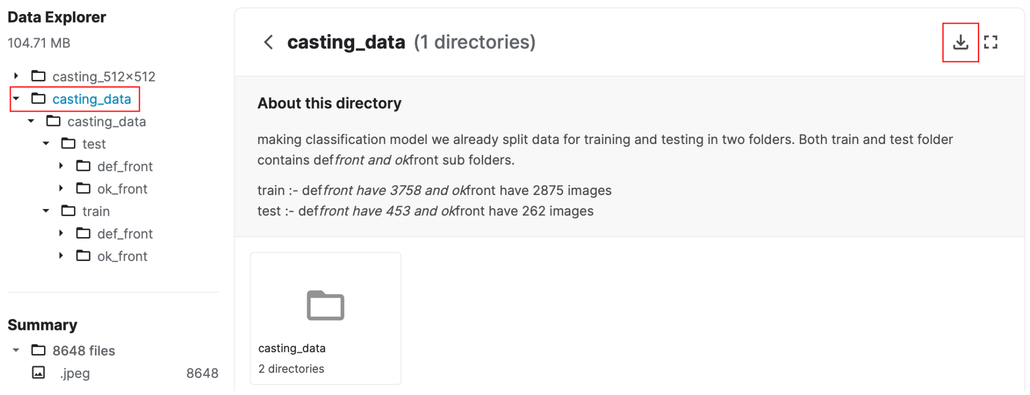
where γ is obtained by first dilating the incoming gradient with size S, and then padding the result with $K-1$ zeros. In other words, this derivative is obtained by computing the convolution of the kernel κ and the image γ.

Question 2.5 Using the explanations above, fill in the `backward` and `update` methods of the `Conv2D` class.

Hint. *Pay attention to the intuitive explanations of the mathematical equations. You already have the heavy operations implemented in the* `convolve`, `dilate`, *and* `add_padding` *functions; you just need to apply them in the correct order to the correct arrays.*
Additionally, notice that Eq. (9) is a convolution with only one input and output filters.

This completes our implementation of a CNN using NumPy. If you play around with this network, you will notice that it is extremely slow at prediction and training. For this reason, we cannot use our implementation on a dataset of reasonable size. In the next section, we will rely on PyTorch instead to build and train a CNN.

Going Further (optional) What follows in this paragraph is not part of the assignment, and is only relevant if you want to go further on this topic. In particular, it is not graded.
One of the main reasons explaining this slowness is that we heavily rely on `for` loops for convolutions and pooling. Loops are notoriously slow in Python, and a good pythonic code should avoid them as much as possible. Here, we could rely on NumPy's very efficient matrix product to speed up our implementation of convolution. To do so, we need to express the convolution operation (1) as a matrix product $Z = K \cdot X$, where Z, K, and X are matrices that organize respectively the sequences $(z_{p,q}^{f_{\text{out}}})$, $(k_{i,j}^{f_{\text{in}},f_{\text{out}}})$, and $(x_{s,t}^{f_{\text{in}}})$ in a smart way. If you are interested in doing this, you can follow the following steps:


Data Explorer

104.71 MB

- casting_512×512
- casting_data
  - casting_data
    - test
      - def_front
      - ok_front
    - train
      - def_front
      - ok_front

Summary

- 8648 files
  - .jpeg 8648

casting_data (1 directories)

About this directory

making classification model we already split data for training and testing in two folders. Both train and test folder contains def*front and ok*front sub folders.

train :- def*front have 3758 and ok*front have 2875 images
test :- def*front have 453 and ok*front have 262 images

casting_data
2 directories

Figure 2: The folder to download and the download button are indicated by the red squares. Source [6].

1. Find a suitable mathematical expression for the matrices Z, K, and X. You will find that you need to reshape all of the sequences $(z^{f_\text{out}}_{p,q})$, $(k^{f_\text{in},f_\text{out}}_{i,j})$, and $(x^{f_\text{in}}_{s,t})$ in the same specific way;
2. Implement a function `im_2_col` implementing this transformation. You may also need a `col_2_im` function implementing the inverse transformation;
3. Implement `convolve_fast`, which computes a convolution as a matrix product
4. Re-implement the `MaxPooling` class without `for` loops using the `im_2_col` function instead.

Once you have done that, your network should be significantly faster and be usable. You can find indications on how to achieve this in [4, 5]

3. Building a Convolutional Neural Network

In this section, we use the state-of-the-art library PyTorch to build and train a convolutional neural network. We will use this network to solve a classification task of a casting dataset [6], where the goal is to identify irregularities in a casting process from pictures.

Question 3.1 Download the data at the following address:

https://www.kaggle.com/ravirajsinh45/real-life-industrial-dataset-of-casting-product

You will need about 100 MB of free space. Choose the `casting_data` folder that is already split into train and test, as described on Fig. 2. Put the downloaded files in the `data` directory, such that your folder architecture is:

```
root
└─ data
   ├─ train
   │  ├─ def_front
   │  └─ ok_front
   └─ test
      ├─ def_front
      └─ ok_front
```

Hint. *If you are using the RWTH Jupyter Hub, Google Colab, ... or anything different than your own computer, you will need to upload the data accordingly so your notebook can access it.*

Question 3.2 Import the training and test data as two `torch.utils.data.Dataset` objects, and store them in `train_ds` and `val_ds`. Plot 5 samples of each class of the training dataset. We will use the test set as a validation set.

Hint. *Use* `torchvision.datasets.ImageFolder` *to load the dataset.*

Question 3.3 As a warm-up task, construct a simple convolutional neural network with PyTorch. A model in PyTorch is a subclass of `torch.nn.Module`. In the `__init__` method, all layers and parameters are created. The `forward` function defines how the network processes its input, similar to the `FeedForwardNet` class from parts 1 and 2. However, it is not necessary to define a `backward` method as PyTorch computes the gradients automatically.
Fill in the `__init__` and `forward` methods of the `SimpLeNet` class to implement and train a convolutional neural network in PyTorch. This model should have:

- A convolutional layer with $K = 3$, $P = 1$, $S = 1$, $F_{\text{out}} = 16$, and ReLU activation, and F_{in} is chosen in accordance with the data we are considering;
- A max-pooling layer with $K = 2$ and $S = 2$;
- A fully connected layer with ReLU activation and 16 neurons;
- A fully connected layer with sigmoid activation and 1 neuron.

Hint. *You can use this webpage for guidance on how to define and train a CNN.*

Question 3.4 Fill in the `train` function. The first part to fill is the creation of a `Dataloader` object that can provide batches of data and shuffles the dataset. The second part is the computation of the validation loss, computed on the `val_ds` dataset.
Use this function to train the `SimpLeNet`.

Hint. *Use PyTorch's documentation and online examples.*

Question 3.5 Change the architecture of the above network and other parameters such as the learning rate or batch size to achieve the highest accuracy you can on the validation set. You are free to add convolutional, fully connected, or pooling layers. You can also apply `torchvision.transforms` to the input data if you wish to.
Your model should have less than 500 000 trainable parameters as indicated by the `train` function. It should also end on a fully connected layer with one neuron, and use the binary cross-entropy loss.
Additionally, you will not be graded based on the performance of your model.

A. Computation of the Gradient w.r.t. Kernel Weights

We prove here the expression of the gradient of the loss w.r.t. the other kernel weights given in Eq. (9).

The chain rule applied to the gradient of the loss w.r.t. the kernel weight with index $(i, j, f_{\text{in}}, f_{\text{out}}) \in \{0, \dots, K-1\}^2 \times \{0, \dots, F_{\text{in}} - 1\} \times \{0, \dots, F_{\text{out}}\}$ yields:

$$\begin{aligned} \frac{\partial L_n}{\partial k_{i,j}^{f_{\text{in}}, f_{\text{out}}}} &= \sum_{p=0}^{h_{\text{out}}-1} \sum_{q=0}^{w_{\text{out}}-1} \frac{\partial L_n}{\partial z_{p,q}^{f_{\text{out}}}} \cdot \frac{\partial z_{p,q}^{f_{\text{out}}}}{\partial k_{i,j}^{f_{\text{in}}, f_{\text{out}}}} \\ &= \sum_{p=0}^{h_{\text{out}}-1} \sum_{q=0}^{w_{\text{out}}-1} \frac{\partial L_n}{\partial z_{p,q}^{f_{\text{out}}}} \cdot \left(\sum_{s=0}^{h_{\text{in}}-1} \sum_{t=0}^{w_{\text{in}}-1} x_{s,t}^{f_{\text{in}}, f_{\text{out}}} \cdot \mathbb{1}_{0,0}(s - p \cdot S - i, t - q \cdot S - j) \right), \end{aligned} \tag{13}$$

where $\mathbb{1}_{i,j}(u, v)$ is the indicator function defined as:

$$\mathbb{1}_{0,0}(u, v) = \begin{cases} 1, & \text{if } u = 0 \text{ and } v = 0, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

We change the order of the sums to sum over the variable s and t first, and define:

$$g_{u,v}^{f_{\text{out}}} = \sum_{p=0}^{h_{\text{out}}-1} \sum_{q=0}^{w_{\text{out}}-1} \frac{\partial L_n}{\partial z_{p,q}^{f_{\text{out}}}} \cdot \mathbb{1}_{0,0}(u - p \cdot S, v - q \cdot S). \tag{15}$$

With this notation, we can rewrite the chain rule equation (13) as:

$$\frac{\partial L_n}{\partial k_{i,j}^{f_{\text{in}}, f_{\text{out}}}} = \sum_{s=0}^{h_{\text{in}}-1} \sum_{t=0}^{w_{\text{in}}-1} g_{s-i,t-j}^{f_{\text{out}}} \cdot x_{s,t}^{f_{\text{in}}, f_{\text{out}}}. \tag{16}$$

This shows that $\frac{\partial L_n}{\partial k_{i,j}^{f_{\text{in}}, f_{\text{out}}}}$ is the convolution between the kernel $g^{f_{\text{out}}}$ and the input $x^{f_{\text{in}}}$. We also have the following simpler expression for $g_{s,t}^{f_{\text{out}}}$:

$$g_{s,t}^{f_{\text{out}}} = \begin{cases} \frac{\partial L_n}{\partial z_{\frac{s}{S}, \frac{t}{S}}}, & \text{if } \frac{s}{S} \text{ and } \frac{t}{S} \text{ are nonnegative integers,} \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

This matrix g is the dilation of $\frac{\partial L_n}{\partial z_{p,q}}$ with a dilation size of S, which concludes the proof.

References

[1] Wikipedia, "Kernels (image processing)," https://en.wikipedia.org/wiki/Kernel_(image_processing), webpage visited on Jan. 29th, 2022.

[2] M. Kaushik, "Part 2: Backpropagation for convolution with strides," https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-fb2f2efc4faa, 2019, webpage visited on Jan. 27th, 2022.

[3] ——, “Part 1: Backpropagation for convolution with strides,” https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710, 2019, webpage visited on Jan. 27th, 2022.

[4] S. Sekhar, “Implementing convolution without for loops in numpy!!!” https://medium.com/analytics-vidhya/implementing-convolution-without-for-loops-in-numpy-ce111322a7cd, 2020, webpage visited on Jan. 29th, 2022.

[5] neuron whisperer, “cnn–numpy,” https://github.com/neuron-whisperer/cnn-numpy, 2020, webpage visited on Jan. 29th, 2022.

[6] R. Dabhi, “Casting product image data for quality inspection,” https://www.kaggle.com/ravirajsinh45/real-life-industrial-dataset-of-casting-product, webpage visited on Jan. 29th, 2022.