Deep Learning for DataMatrix Code Recognition

Vanpeene Paul

Research Internship: May 19 - September 14, 2025

Catholic University of Louvain
ENSEIRB-MATMECA - 2nd Year

GitHub Repository Access

All code described in this report is available in this repository

`github.com/vanpeenepaul`

⚠ **Throughout this report, "(-)" symbols redirect to a source or research paper, "(*)" to theoretical elaboration, "(°)" to other previous sections of the report, "(§)" to Python library descriptions, and finally "(.)" to a definition.**

Contents

1 Context and Research Problem Statement

1.1 Working Environment at Catholic University of Louvain

I conducted four months of research at the Catholic University of Louvain (UCLouvain), located in Louvain-la-Neuve, Belgium. Founded in 1425, UCLouvain is the most recognized French-speaking university in Belgium (-). It is distinguished by the quality of its teaching and research, particularly in science and technology, but also in humanities and social sciences. Its Louvain-la-Neuve campus hosts a large international community of students and researchers.

I conducted my work in the Maxwell building, located at Place du Levant 3, 1348 Ottignies-Louvain-la-Neuve, within the deep learning research department. I was able to join this laboratory thanks to Professor Christophe De Vleeschouwer, my supervisor, who oversees researchers at the university. My direct supervision was provided by Sarra Laksaci, a doctoral student, with whom I collaborated daily.

Sarra's research is funded as part of a Win4Doc project, supported by the Walloon Region. This project results from collaboration between UCLouvain and the company Euresys, which defined the research topic. Sarra officially began her thesis in February 2024, with the objective of developing **a new solution for reading Data Matrix codes using deep learning-based computer vision methods**. Her work is part of the exploration and improvement of advanced deep learning techniques, aimed at optimizing interpretation and strengthening the robustness of automatic reading of these codes.

It is on one of the learning methods developed as part of her thesis that my own research work, presented in this report, is based.

During the first month, I had daily meetings with Sarra to understand the challenges of my research topic and receive the necessary scientific articles to identify the elements to implement. For the following three months, I worked more independently.

1.2 Research Domain and Thematic Orientation

The subject I studied concerned DataMatrix codes. DataMatrix codes are two-dimensional codes composed of black and white square modules (**cells**) arranged in a regular geometric pattern. These codes are widely used in industry for product identification and traceability, particularly in the automotive, pharmaceutical, and electronics sectors, due to their ability to store a large amount of information in a limited space and their resistance to damage.

However, my subject did not focus on encoding or understanding DataMatrix codes. **The goal of my work was to implement a deep learning method to predict the position of cells in a DataMatrix code**. In industry, DataMatrix codes are widely used, but a DataMatrix code can be damaged, engraved on a deformed or curved material, or the photo quality of the code may be poor. In these cases, traditional code detectors struggle to function. Below are difficult-to-decode DataMatrix codes that were present in the dataset (.) on which I worked.

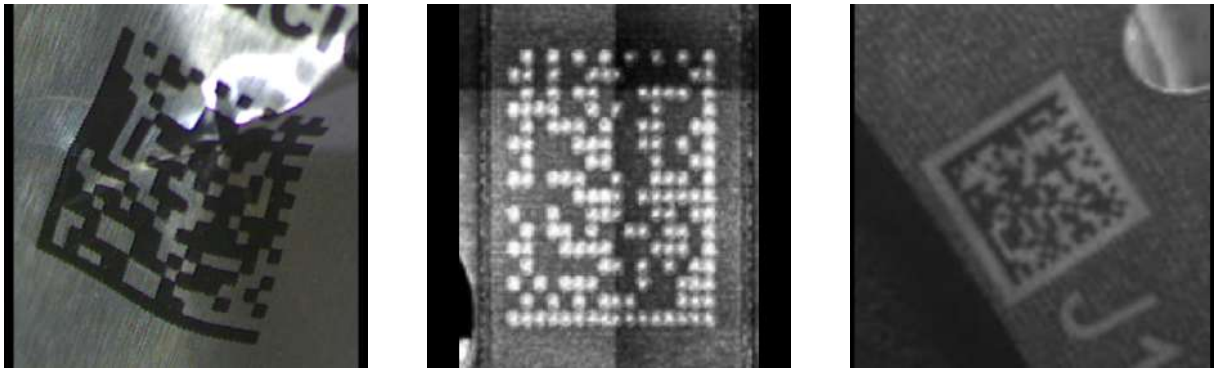

Figure 1: DataMatrix codes difficult to read due to degradation.

The specific goal of my study was thus to predict the cells of a DataMatrix code regardless

of its condition and regardless of the quality of the photo in which it appears. Thus, once the position of cells in a DataMatrix code, even damaged, is predicted, they could then be processed to be readable by a DataMatrix code reader, but this is no longer part of what I study. I focus on how to predict the position of cells in a DataMatrix code.

The theme of this internship fits within the deepening of the course "Artificial Intelligence for Image Processing" by Rémi Giraud that I took at ENSEIRB in the signal processing specialty.

To carry out my research, bottom-up and top-down approaches could be implemented. The bottom-up approach first analyzes basic elements (pixels, contours) to progressively reconstruct the global structure, offering great robustness to deformations and occlusions but at the cost of high computational complexity. Conversely, the top-down approach directly searches for characteristic DataMatrix patterns in the global image, ensuring faster detection and less prone to false positives, but less flexible to quality or deformation variations. Thus both methods could be considered, but to specifically address the problem of my subject, I was required to implement a bottom-up method.

1.3 Description of Our Specifications

My specifications for this internship consisted of implementing deep learning training in a bottom-up approach, based on a pre-trained convolutional neural network architecture (.). The resulting model had to be capable of predicting the position and class (black or white) of cells in any DataMatrix code present in an image, regardless of its viewing angle, geometric deformations, as well as the image quality and colorimetry. My training had to additionally have two modes. A unified mode seeking to predict the position of all DataMatrix code cells, whether black or white. A non-unified mode seeking to separately predict the position of black cells and white cells. To accomplish this task, I had a dataset composed of 528 images distributed across 6 distinct folders, each folder containing annotation files (.) in JSON format. An additional constraint of my specifications required exclusively manipulating annotation files in COCO JSON format (.) during my training, the COCO annotation method being the most common in this research domain. Finally, the last constraint required using the PyTorch framework for model development.

Finally, the two final objectives of this internship were on one hand an engineering objective: create an open source GitHub repository that allows anyone with a DataMatrix dataset to reproduce my experiments and use the deep learning model for DataMatrix code detection that I developed; on the other hand a research objective: experiment with different techniques and compare them to best predict the cells of a DataMatrix code, whatever its condition.

2 Development of Python Scripts

2.1 Section Description

This section presents the different notebooks available in my GitHub repository, which you can consult alongside your reading. These notebooks correspond to the final code developed during this internship. In the titles of this section, elements in `dark blue` indicate the names of the studied notebook files, while the notation cell_i refers to cell number i of the corresponding notebook.

2.2 Data Visualization Script: `Visualization`

Throughout my study, visualization of manipulated data is essential to understanding the implemented processes. This is why I developed a script dedicated to visualization, and all graphical representations presented hereafter were generated thanks to it.

2.3 Dataset Preprocessing and Formatting Script: `Conversion dataset`

The objective of this subsection was to standardize the dataset on which I worked, so that it corresponds to the intended objective, namely using annotations in COCO format.

The dataset I had at the beginning of the internship contained two annotation files: the first was a classic dictionary separately indicating, in a binary grid, the position of black and white cells of a DataMatrix code; the second was in COCO format, but only indicated the position of cells, without class distinction (black or white).

First, I realized that, for certain images, a 0 in the binary grid represented the position of a black cell, while for others, this same 0 could correspond to a white cell. The annotator had not standardized the meaning of 0s and 1s in the binary grid. My first task was therefore to normalize this representation. To do this, I relied on the L property of DataMatrix codes (see diagram): at least two consecutive sides of the DataMatrix contain only black cells. Otherwise, I inverted the code. At the end of this process, each 0 corresponded to a white cell and each 1 to a black cell. (`cell5+6`)

Figure 2: Property of L.

The second work consisted of using the dictionary equipped with this binary grid to modify the COCO format file, so that it also integrates class distinction. (`cell5+6`)

Finally, in an open source perspective, it seemed important to me to add a final code allowing, from a dataset gathered in a single folder containing images and their annotations, to generate a dataset compliant with our project (`cell7`). This code automatically segments a collection of images into visually similar groups using the K-Means algorithm from sklearn.cluster (§), applied to colorimetric statistics (means and standard deviations of RGB channels). It then distributes the corresponding COCO annotations by producing, for each cluster, a distinct JSON file containing only the metadata of images in this group.

Thus, at this stage, the positions and classes (black or white) of DataMatrix present in the image were recorded in an annotation file in COCO format. Here is the representation:

This subsection allowed me to progress in data manipulation in Python, particularly through file and

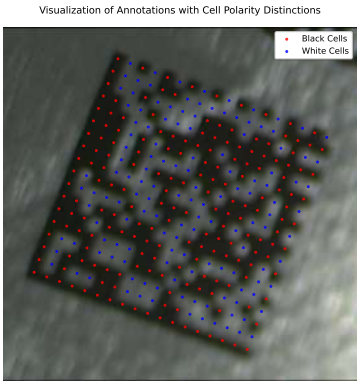

Figure 3: Annotation/image overlay.

dictionary management. It also helped me design more sophisticated Python structures than traditional structures, such as with the use of classes.

2.4 Dataset Augmentation Script: `Transformations`

I wanted my model to be able to predict the position of DataMatrix cells even in complex conditions, related to shooting or suffered degradations. My model therefore had to train on images representing these cases. The objective of this subsection was thus to augment my dataset with such images, such transformations being called offline (outside of training).

First, to reproduce the unconventional shooting and colorimetry observed on certain images, I applied **elementary geometric transformations** (rotations at a random angle) as well as **colorimetric transformations** (Gaussian noise and contrast modification) to the data. All these transformations were performed with the Albumentations library (§) and resulted in a dataset four times larger. The annotations of new images were then added to our annotation file `cell3+4`. Here is a visualization of the results:

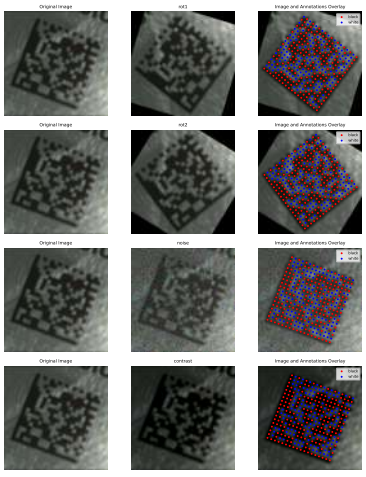

Figure 4: Offline transformations with Albumentations

Next, the two other types of deformations I observed in my dataset images correspond to cases where the **DataMatrix is curved or crumpled**, as illustrated below:

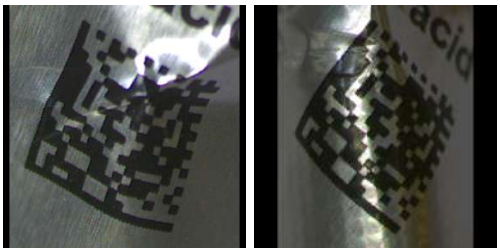

Figure 5: Crumpled/curved DataMatrix code.

I thus had to find a way to implement a method to curve the images and annotations of DataMatrix in my dataset, by applying a sinusoidal transformation along the vertical axis. Each pixel is displaced horizontally based on its vertical position, and the coordinates of keypoints (.) are recalculated according to the same transformation to preserve consistency between the modified image and its annotations. Key points are also adjusted to remain within image boundaries `cell6`. This transformation is detailed more in depth here: (*). Here is the visual representation:

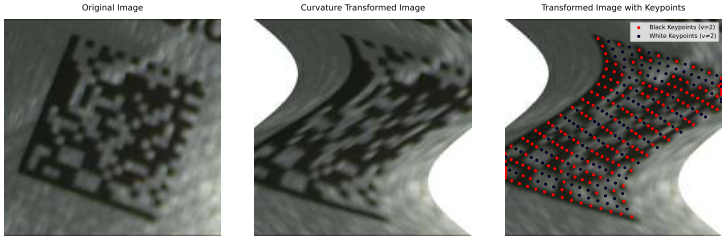

Figure 6: My curvature transformation.

Finally, the final objective was the crumpling transformation. I first tried to implement a method with Perlin noise. (.) However, Perlin noise generates a continuous deformation field over the entire image, but keypoints constitute a discrete set of isolated points in space, and therefore I did not succeed in adapting Perlin noise to our study.

I finally drew inspiration from the previous transformation to create a local curvature, that is for each pixel of my image, in a random manner. For the keypoint transformation, I approximated their transformations using the transformations of the 4 neighboring pixels at integer coordinates. `cell7` This transformation is detailed more in depth here: (*).

I adapted all my transformation parameters manually by visualizing the annotation-image overlay. The final transformation is subtle, because by increasing the intensity of this transformation, the consistency of the overlay between the image and its keypoints is no longer respected. Here is a limit case where the crumpling transformation is visible and where the annotation/image overlay is about to no longer be valid.

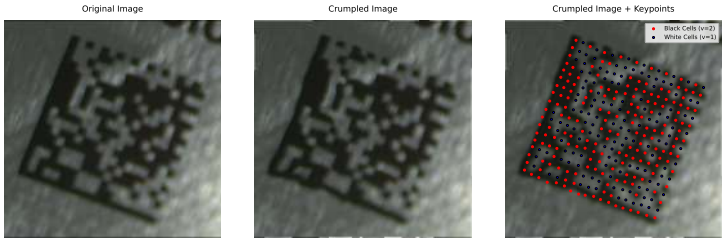

Figure 7: My crumpling transformation.

Finally, I applied these two transformations to my dataset, updating the annotation file as previously with offline Albumentations transformations, which multiplied my dataset size by 3.

This subsection allowed me to really progress in implementing mathematical functions to transform an image.

2.5 Training Dataset Formation Script: `Training dataset`

The objective of this subsection was to adapt our 6-subfolder dataset to a format conducive to training. The idea was to apply the K-fold method: we separated our 6 subfolders into 2 sets. The first contained 5 of the 6 folders with an annotation file of corresponding images: this is the set on which our model will train. The second contained only one folder of the 6 with the associated annotation file: this is the set that will allow us to evaluate our model's performance. Indeed, it would make no sense to evaluate our model on images on which it trained, the model could very well be overfitting (.).

Moreover, an even more complete method to verify performance would be to construct all possible combinations of subfolders: 5 folders for training and the remainder for evaluation, to verify that our model performs independently of training data, as long as they present the degradations on which we seek to detect the position of DataMatrix code cells.

2.6 Deep Learning Training Pipeline Script: `Training`

2.6.1 Creation of Target Heatmaps `cell2`

First, for each image in our dataset, I had to create target heatmaps representing the position of DataMatrix code cells present in the image. These positions were contained in the COCO JSON files. These are the target heatmaps that our model will have to be able to create by itself for each image in our dataset.

First, COCO annotations and raw images must be loaded then resized to target resolution $(m, n) = (512, 512)$. Coordinates are scaled according to the formula $(x', y') = \left(x \cdot \frac{n}{W}, y \cdot \frac{m}{H}\right)$ to remain consistent with the new image size.

Target heatmap generation From resized keypoints, **we construct continuous heatmaps** $T_{b,c} \in [0,1]^{m \times n}$ **(where** $b \in \{1, \ldots, B\}$ **designates the image index in the batch of size** B **and** c **designates the output channel corresponding to cell type)** by placing, on each point (x', y'), an isotropic Gaussian with standard deviation σ:

$$T_{b,c}(i,j) = \max_{k \in \mathcal{K}_c} \exp\left(-\frac{(i-y'_k)^2 + (j-x'_k)^2}{2\sigma^2}\right), \qquad (i,j) \in \Omega, \quad |\Omega| = mn$$

$$\mathcal{K}_c = \{k : \text{keypoint } k \text{ belongs to class } c \text{ in the image}\}$$

In **unified mode** ($C = 1$), all cells feed the same map. In **separated mode** ($C = 2$), we produce two channels: $c = 1$ (black cells) and $c = 2$ (white cells).

Here is a representation of the target heatmaps thus produced:

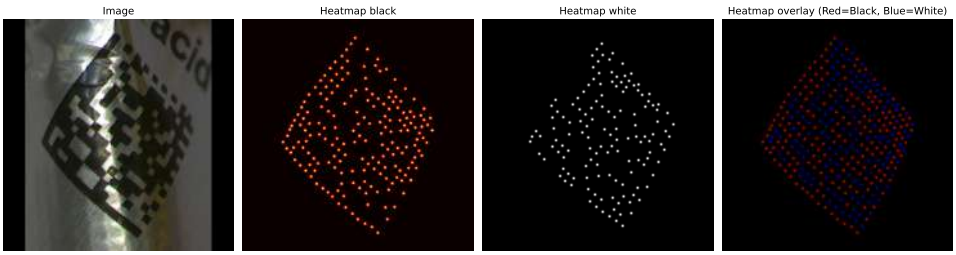

Figure 8: Visualization of target heatmaps

Online transformations. A trick to significantly reduce overfitting (.) consists of applying data transformations in real time during training, rather than loading pre-transformed images. Thus, during training, the model never visualizes exactly the same images from one epoch to another. My curvature

and crumpling transformations were too computationally expensive to be executed online, however, more elementary geometric transformations and colorimetric transformations can be performed in real time thanks to the Albumentations library. Thus we apply transformations:

- **Geometric**: rotations, moderate translations/scale, horizontal/vertical flip. And keypoints undergo exactly the same transformations (`keypoint_params`).
- **Photometric**: brightness/contrast variation, color jitter (hue/saturation/value), gamma, noise (Gaussian/ISO), blur (motion/Gaussian).

Here are examples of images having undergone these transformations:

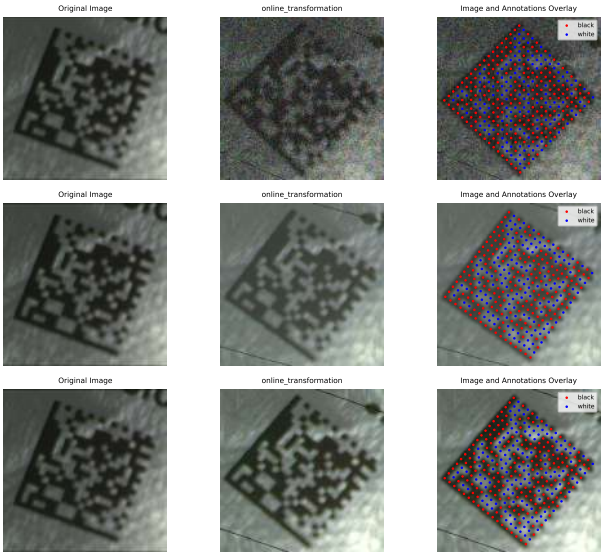

Figure 9: DataMatrix codes difficult to read due to degradations suffered.

2.6.2 Neural Network Architecture Implementation `cell3`

The idea of this part was to develop a function that associates heatmaps to an image that predict the positions of DataMatrix code cells.

The way I proceeded to build my CNN was greatly inspired by the U-Net architecture (-). The idea consisted of using a model already trained on object detection in an image (called backbone) and building our predicted heatmaps from this model. The backbone weights being already trained on numerous data, this facilitates our training predictions rather than initializing it with random weights.

After several unsuccessful attempts (often due to version compatibility issues) with ICNet and HRNet models, I finally chose to base my backbone on the use of pre-trained `ResNet` networks (-) from the `torchvision` library (`resnet18`, `resnet34` or `resnet50`). In each case, the final layers dedicated to classification are removed and replaced by a truncated feature extractor, keeping only the convolutional layers. Thus, the dimension of produced tensors is 512 channels for `ResNet-18/34` and 2048 channels for `ResNet-50`. These feature maps are then passed to a decoder module consisting of a series of transposed convolutions (`ConvTranspose2d`) that ensure progressive upsampling of resolution: $256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16$ channels. Each block is followed by batch normalization (`BatchNorm2d`), non-linear `ReLU` activation, and `Dropout2d` (for the first three blocks) to limit overfitting. Finally, a 3×3 convolution generates the network's final output, whose number of channels is set to 1 (unified mode) or 2 (separated black/white mode). A `Sigmoid` function is finally applied to constrain predictions between 0 and 1. Here is a summary of the implementation of our architecture:

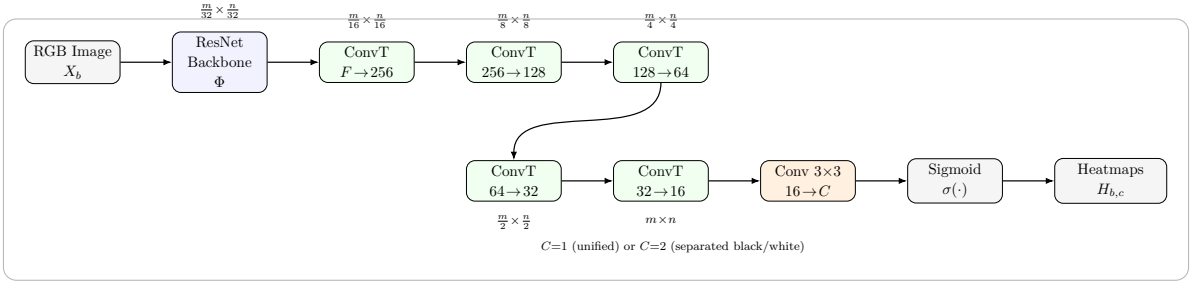

Figure 10: Network architecture: ResNet backbone for feature extraction then decoder by progressive upsampling to resolution $m \times n$.

And here is the visualization of the predicted heatmap for an input image before the start of our training:

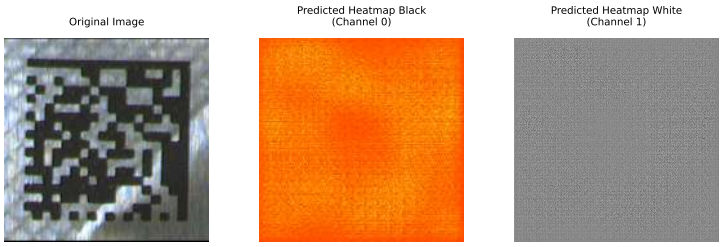

Figure 11: Visualization of predicted heatmaps before training

2.6.3 Design and Implementation of Loss Function `cell4`

The challenge of this section was to implement the function that will determine how close the target and predicted heatmaps are. It is this function that we will seek to minimize during our training.

MSE Loss. I started by implementing the following MSE loss:

$$L_{\text{MSE}} = \frac{1}{B}\sum_{b=1}^{B} \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} \begin{cases} \big(H_{b,1}(i,j) - T_{b,1}(i,j)\big)^2, & \text{(unified, } C=1\text{)}, \\ \displaystyle\sum_{c=1}^{2} \big(H_{b,c}(i,j) - T_{b,c}(i,j)\big)^2, & \text{(separated, } C=2\text{)}. \end{cases}$$

where: $H_{b,c} \in [0,1]^{m\times n}$ is the predicted heatmap.
This loss function is the most commonly used, however it treats all pixels equally, which can lead the model to favor predicting the background rather than precisely detecting cells. Hence the need to implement the following loss function:

Focal Loss. First we define pixel-by-pixel BCE (without reduction) by:

$$\ell_{\text{BCE}}(p,t) = -\big[t\log p + (1-t)\log(1-p)\big], \quad p \in (0,1) \text{ prediction}, \quad t \in \{0,1\} \text{ target}$$

Then we set $p_t = \exp\big(-\ell_{\text{BCE}}(p,t)\big) = p^{\,t}(1-p)^{\,1-t}$, then the focal loss

$$\ell_{\text{focal}}(p,t) \;=\; \zeta\,\big(1-p_t\big)^{\eta}\,\ell_{\text{BCE}}(p,t), \quad \text{with } \zeta,\ \eta \text{ hyperparameters.}$$

(.) Let's now visualize the advantage of combining MSE and Focal losses:

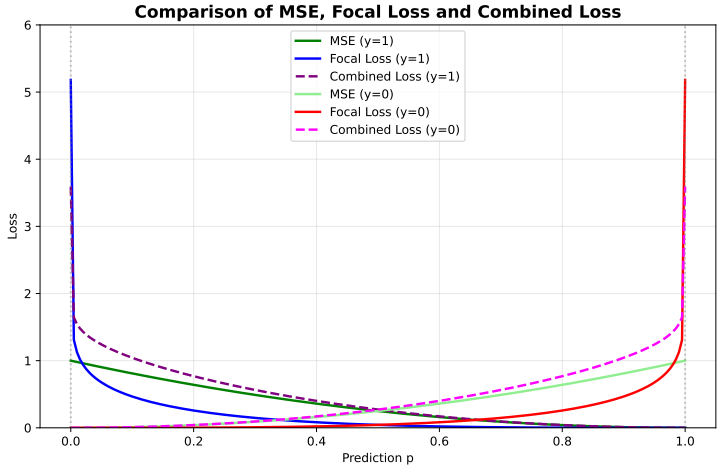

Figure 12: Visualization of MSE and Focal loss function combination

According to this graph, Focal Loss focuses on significant errors: it ignores correct predictions (low loss at center) but heavily penalizes large errors (peaks on sides). For DataMatrix, this prevents the model from neglecting rare cell pixels in favor of the majority background.

This function was then calculated at image and batch scale in the same way as for L_{MSE}.

I finally implemented a **Contrast Loss** composed of two "pull" and "push" terms. My approach is inspired by the associative embedding method (-)., but does not constitute AE in the strict sense: I do not learn embedding vectors per instance and do not use a head dedicated to tags. My loss function operates only on the scalar intensity of heatmaps.
The calculations performed in my code rely on matrix operations equivalent to those that follow, but less practical to present.

My contrast loss is composed of two terms: L_{pull} and L_{push}. Let $H = (h_{ij}) \in \mathbb{R}^{m\times n}$ be the heatmap predicted by the model and $G = (g_{ij}) \in \{0,1\}^{m\times n}$ the binary target map defined by:

$$g_{ij} = \begin{cases} 1 & \text{if } T_{ij} > \tau \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where T represents a predicted heatmap and τ is a thresholding hyperparameter.
In our case, we have $m = n = 512$, i.e. maps of size 512×512.
I then defined the index sets

$$\Omega_1 = \{(i,j) \mid g_{ij} = 1\}, \qquad \Omega_0 = \{(i,j) \mid g_{ij} = 0\}.$$

Then set the activation averages per group:

$$\mu_1 = \frac{1}{|\Omega_1|} \sum_{(i,j)\in\Omega_1} h_{ij}, \qquad \mu_0 = \frac{1}{|\Omega_0|} \sum_{(i,j)\in\Omega_0} h_{ij}.$$

My **pull** term then corresponds to intra-group variance:

$$L_{\text{pull}} = \frac{1}{2|\Omega_1|} \sum_{(i,j)\in\Omega_1} (h_{ij} - \mu_1)^2 + \frac{1}{2|\Omega_0|} \sum_{(i,j)\in\Omega_0} (h_{ij} - \mu_0)^2,$$

which forces activations of the same group to approach their mean.
My **push** term acts oppositely and favors separation between groups:

$$L_{\text{push}} = \frac{1}{|\Omega_1| + |\Omega_0|} \left(\sum_{(i,j)\in\Omega_1} \max\left(0,\, 1 - (\mu_0 - (h_{ij}))^2\right) + \sum_{(i,j)\in\Omega_0} \max\left(0,\, 1 - (\mu_1 - (h_{ij}))^2\right) \right).$$

Thus, minimizing L_{pull} reduced intra-class dispersion, while minimizing L_{push} increased inter-class separability. This function encouraged both *compactness* of predictions per class and their *discriminability*.

Combined losses. Finally my final loss function was:

$$L_{\text{total}}^{(\text{MSE+Focal+Contrast})} = \alpha\, L_{\text{MSE}} + \beta\, L_{\text{focal}} + \gamma\, L_{\text{push}} + \delta\, L_{\text{pull}},$$

with α, β, γ, δ, hyperparameters.

2.6.4 Performance Evaluation Metrics `cell5`

To evaluate our model's prediction performance, I implemented several complementary evaluation metrics. The MSE loss function serves as a standardized validation metric, independent of the training loss function used (L_{MSE}, $L_{\text{MSE}} + L_{\text{focal}}$, or $L_{\text{MSE}} + L_{\text{focal}} + L_{\text{contrast}}$).

I also implemented Average Precision (AP) from `sklearn.metrics`, which measures detection quality by comparing predicted heatmaps to binarized ground truths. This metric is particularly suitable for evaluating the model's ability to precisely locate DataMatrix code cells. Indeed, while MSE loss evaluates the overall gap between predictions and ground truths, it treats the problem as continuous regression and can be dominated by majority background pixels. Average Precision (AP) complements this approach by specifically evaluating binary detection capability of cells after heatmap binarization. A model can have low MSE thanks to good background prediction but mediocre AP revealing cell localization difficulties, hence the importance of these two complementary metrics.

I finally implemented the PCK (Percentage of Correct Keypoints) metric which evaluates the localization accuracy of keypoints extracted from heatmaps. It calculates the percentage of predicted keypoints located within a tolerance radius (default 20 pixels) relative to reference keypoints. This metric is calculated separately for black and white cells in separated mode, or for all cells in unified mode. (*).

Finally, I developed comprehensive visualization functions that automatically generate detailed graphs to track the evolution of all these metrics during training, allowing in-depth analysis of model performance.

2.6.5 Training Auxiliary Functions `cell6+8`

I finally developed several auxiliary functions essential to the training process. The first manages automatic saving of model weights at regular intervals and keeps the best model based on MSE validation loss.

And finally, I designed an interactive configuration interface that automates experimentation parameterization. This function guides the user in selecting four key components: backbone architecture (ResNet-18/34/50), cell detection mode (separated or unified), loss function (MSE, MSE+Focal, or MSE+Focal+Contrast), and data augmentation activation.

2.6.6 Complete Training Workflow `cell8`

The training workflow is based on the `CONFIG` configuration that combines parameters selected by the user via the interactive interface (backbone, loss_type, use_augmentation, unified_cells) with predefined hyperparameters (batch_size, learning_rate (.), weight_decay (.), epochs, etc.).

At each iteration, the model receives mini-batches of normalized images and their targets $T_{b,c} \in [0,1]^{m\times n}$, and produces maps $H_{b,c} \in [0,1]^{m\times n}$ (after `Sigmoid`), with B the batch size, $C \in \{1,2\}$ (unified or separated mode) and $(m,n) = (512, 512)$.

Training loop per epoch. For each epoch $e = 1, \ldots, E$ (with E=80 by default):

1. **Train phase.** For each mini-batch:
   (a) *Online loading & preprocessing* via `DataLoader` (°): resizing to 512×512, generation of target heatmaps $T_{b,c}$ with σ=2, and (in training) geometric/photometric augmentations applied *synchronously* to images and keypoints.
   (b) *Forward pass* of network $X_b \mapsto H_{b,c}$.

(c) *Loss calculation* according to `loss_type`:

$$L_{\text{total}} = \begin{cases} L_{\text{MSE}}, & \texttt{mse}, \\ \alpha\, L_{\text{MSE}} + \beta\, L_{\text{focal}}, & \texttt{mse\_focal}, \\ \alpha\, L_{\text{MSE}} + \beta\, L_{\text{focal}} + \gamma\, L_{\text{push}} + \delta\, L_{\text{pull}}, & \texttt{mse\_focal\_contrast}, \end{cases}$$

L_{MSE} and L_{focal} aggregate over pixels $(i, j) \in \Omega$ then over batch $b \in \{1, \ldots, B\}$, summing channels when C=2. The *contrast* terms L_{push} and L_{pull} operate on heatmap intensity masked by classes (cell/background or black/white).

(d) *Backpropagation & update*: gradient zeroing, `loss.backward()`, then `AdamW` optimization step. Thus, we update our model weights to minimize our loss function.

2. **Validation phase.** The model switches to `eval` mode and we calculate *only* validation MSE (model selection metric). In parallel, we log AP (channel average) and PCK (per channel in separated mode, unique in unified mode).
3. **LR schedule.** `ReduceLROnPlateau` updates learning rate based on validation MSE.
4. **Saves & visualization.**
   - Best model saved *as soon as* validation MSE decreases (`best_model_epoch_***.pth`).
   - Complete checkpoints every 10 epochs.
   - Prediction visualization (GT/prediction overlay) at epoch 1 then every 5 epochs.
   - Curve plots (train/val loss, AP, PCK, and if applicable $L_{\text{push}}/L_{\text{pull}}$) every 10 epochs.

Energy monitoring An asynchronous thread queries `nvidia-smi` every 5 s to estimate average power, energy (kWh) and carbon footprint (Belgium factor 0.11 kg CO_2/kWh). A report and `csv` are produced at end of training. (*)

This training process is summarized by this diagram:

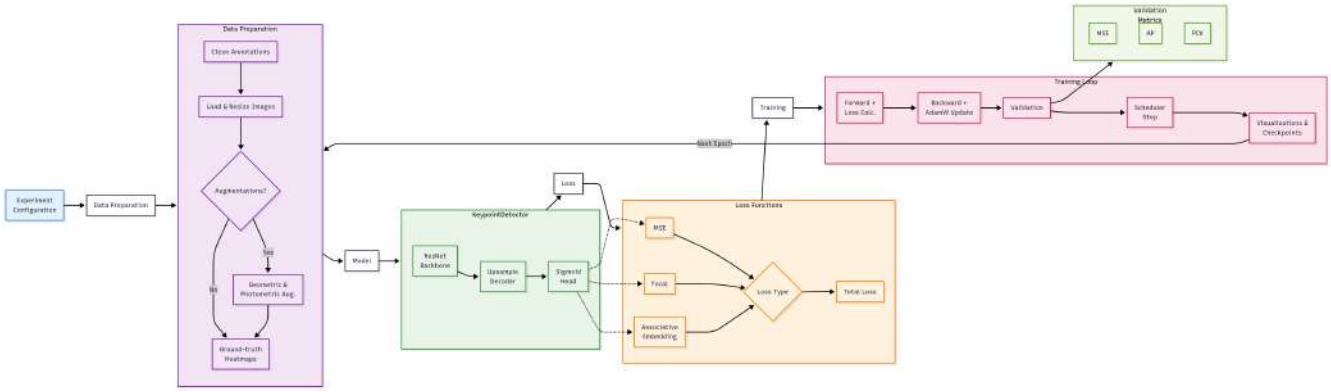

Figure 13: Training process

2.7 Pre-trained Model Loading and Inference Script: `'Demo'`

I finally created a final script allowing to load my model weights once trained, and to predict cells of any DataMatrix code with result visualization.

3 Experimental Results and Performance Analysis

3.1 Experimental Methodology and Hyperparameter Configuration

This section presents interesting results among the many training runs I conducted to determine the optimal method for predicting the position of DataMatrix code cells, including when damaged. Each training involves optimizing different hyperparameters organized according to four main categories.

Data and preprocessing:

- Heatmap generation: Gaussian standard deviation σ
- Online data augmentation (geometric and photometric) or not
- Probabilities of online applied transformations
- Annotation mode: `unified_cells` $\in \{\text{False}, \text{True}\}$ (black/white separation or unified annotation)

Architecture and segmentation head:

- Backbone architecture: `ResNet` $\in \{$`resnet18`, `resnet34`, `resnet50`$\}$

Loss functions and weightings:

- Loss type: `loss_type` $\in \{$`mse`, `mse_focal`, `mse_focal_contrast`$\}$
- Combination weights (`loss_weights`):
  - α: MSE weighting
  - β: Focal weighting
  - γ: *push* weighting
  - δ: *pull* weighting
- Focal Loss parameters: factor $\zeta = 0.25$, exponent $\eta = 2.0$, BCE without reduction

Optimization and training:

- Optimizer: Learning rate (`learning_rate`) and regularization (`weight_decay`)
- Training parameters: number of epochs (`epochs`) and batch size (`batch_size`)

3.2 Quantitative Results and Comparative Performance Evaluation

First, it should be noted that all results presented above were obtained on a laptop, not on the university cluster. I was therefore limited in resources. Consequently, **I restricted my experiments to ResNet-18 backbone**, I did not exceed 80 epochs (.), and batch size (.) was set to 8 for all experiments, in accordance with the recommendation to use a power of 2 (despite an initial limit of 12). Moreover, all training runs I will present correspond to the best results obtained for a given configuration (choice of loss function, transformations or not, etc.), thanks to selected hyperparameters. I mainly focused on non-unified mode, where we predict both cell position and their class (black or white), because this mode is more difficult to optimize. The idea is that once this mode is correctly optimized, the other operating mode would become much simpler to optimize.

3.2.1 Training 1

First, let's analyze the most basic training I launched. The objective of this training was to serve as a reference, to compare results of other configurations and evaluate improvements brought by different optimizations. This training therefore includes no image transformation and uses the simplest loss function: MSE. Here are the chosen hyperparameters and obtained results:

Parameter	**Value**
Gaussian standard deviation (σ)	2.0
Unified mode	False
Online Transformations	False
Loss type	mse
Weight α (MSE)	1.0
Learning rate	0.001
Weight decay	0.0001
Epochs	80
Batch size	8

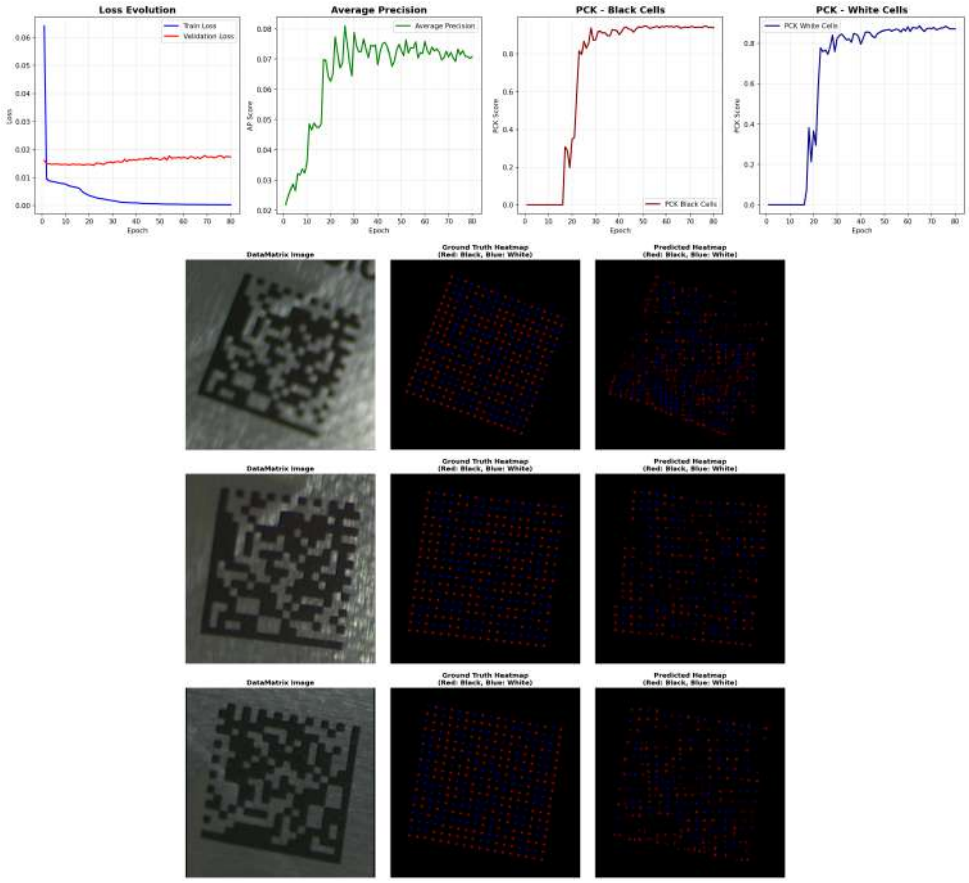

Figure 14: Training 1 results

Thus, as you can see, prediction is poor for simple cases (image 3) and mediocre for complex images with less common colorimetry and rotation (images 2 and 4). Average Precision is low (below 0.1 while the closer AP is to 1, the better the performance). Percentage of Correct Keypoints might seem interesting (around 80

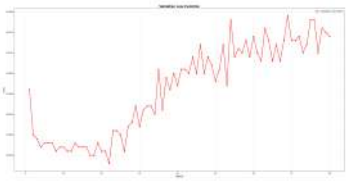

Figure 15: Training 1 validation loss

3.2.2 Training 2

To correct this problem of cell detection in unconventional images, I therefore applied all my offline transformations described here: (°). My dataset size was thus multiplied by 15. Results of this training were unconvincing due to overfitting: although my dataset was richer and presented more variety, the model visualized the same images at each epoch. Moreover, with such a large dataset, training required 4 days, which was not optimal. Nevertheless, curved or crumpled DataMatrix cells seemed to be better predicted.

I then decided to change strategy: **I kept only curvature and crumpling transformations in offline mode (multiplying dataset size by 3), while all other types of transformations were performed online.** With this hybrid approach, my model never sees exactly the same images, which limits overfitting while preserving benefits of complex geometric transformations.

3.2.3 Training 3

Consequently, I trained my model on my dataset containing original images as well as their versions with curvature and crumpling transformations, using MSE loss. Here are the used hyperparameters and obtained results:

Parameter	Value
Gaussian standard deviation (σ)	2.0
Unified mode	False
Online Transformations	True
Online Transformations Probability	0.85
Backbone	resnet18
Loss type	mse_focal_contrast
Weight α (MSE)	1.0
Learning rate	0.001
Weight decay	0.0001
Epochs	80
Batch size	8

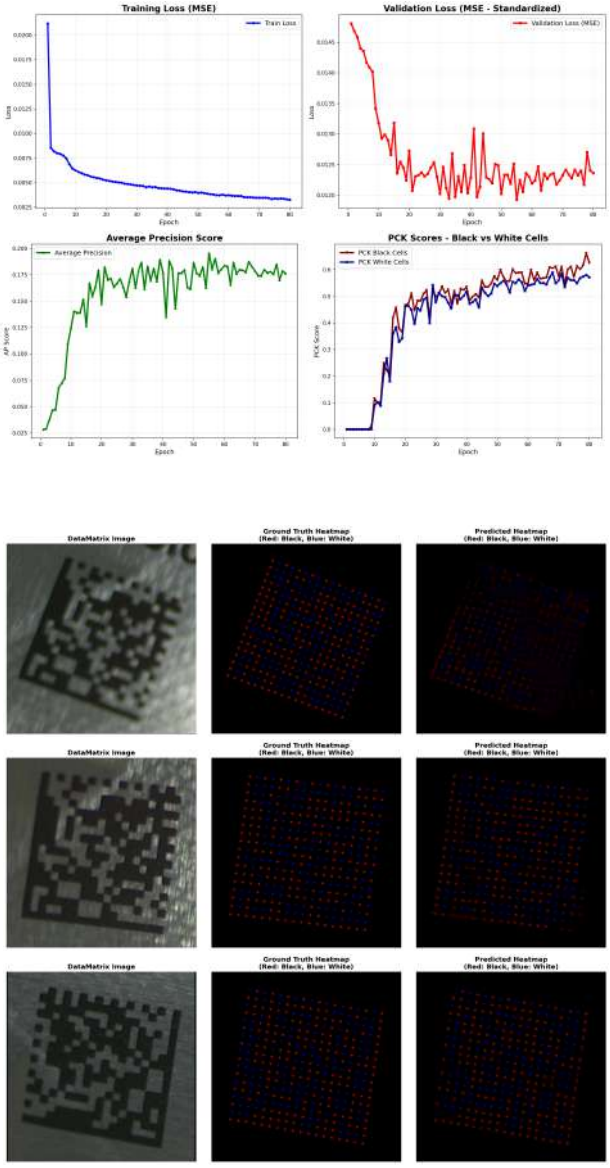

Figure 16: Training 3 results

Thus, visualization images and AP show that this new configuration is superior to the previous one, but remains imperfect. For simple cases (image 3), some keypoints are not predicted, and for more complex cases (image 1), a large number of keypoints are not predicted. Nevertheless, keypoints that are predicted show good localization and predicted heatmaps are much less chaotic than previously. AP remains nevertheless low, PCK is not comparable to that of previous training as mentioned above, and validation loss seems to stagnate, which reflects overfitting (but less important than in previous training as it does not increase either).

3.2.4 Training 4

For the last training, I used the same configuration as previously, but this time with my three combined losses. Here are the used hyperparameters and obtained results:

Parameter	**Value**
Gaussian standard deviation (σ)	2.0
Unified mode	False
Online Transformations	True
Online Transformations Probability	0.85
Backbone	resnet18
Loss type	mse_focal_contrast
Weight α (MSE)	1.0
Weight β (Focal)	0.5
Weight γ (Push)	0.3
Weight δ (Pull)	0.2
ζ (Focal Loss Factor)	0.25
η (Focal Loss Exponent)	2.0
Learning rate	0.001
Weight decay	0.0001
Epochs	80
Batch size	8

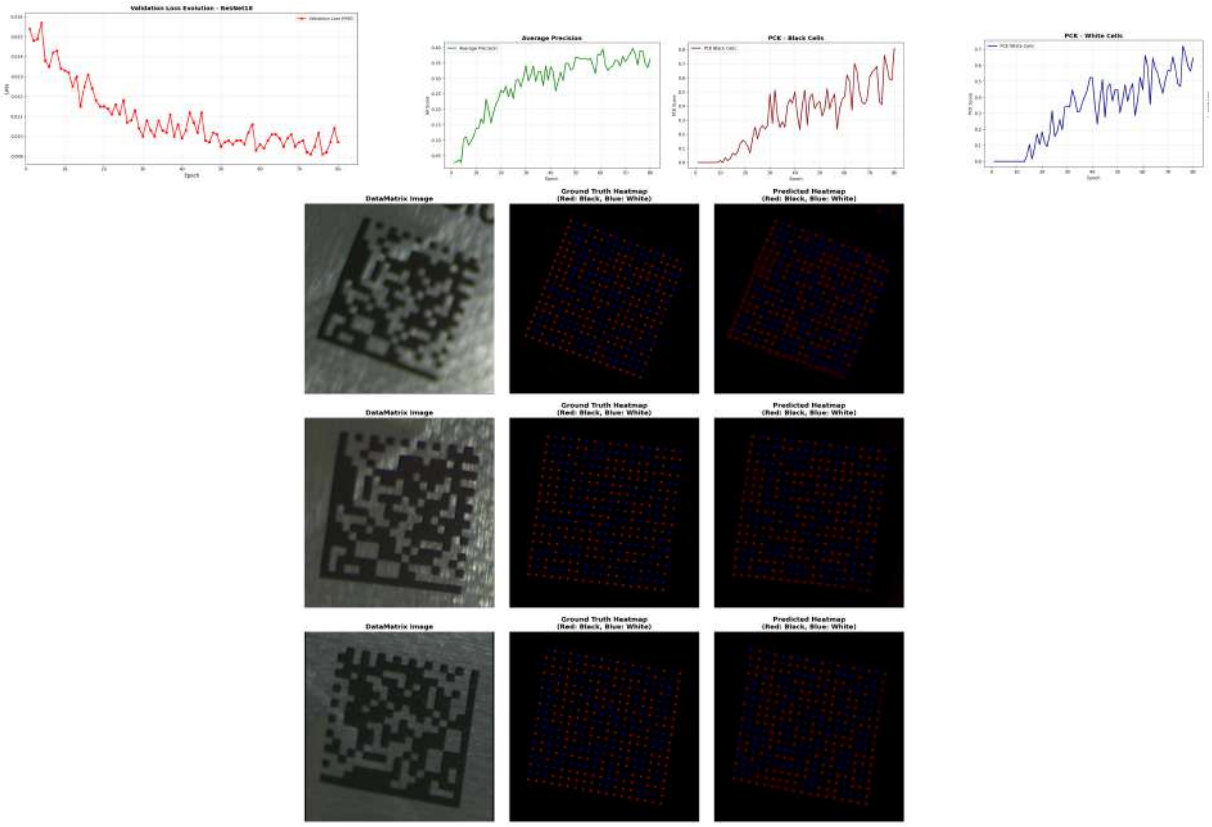

Figure 17: Training 4 results

I then obtain predictions much more satisfactory than with previous training runs. Validation loss globally decreases throughout training, indicating absence of overfitting. AP and PCK are much better, and predictions for images presenting rotation or flip are very satisfactory (images 2 and 3). For more delicate images (image 1), prediction is less good but remains interesting: it should not be forgotten that predicted heatmap corresponds to a probability map. Thus, colored points correspond to zones around cell centers. The study objective being to determine cell center positions (i.e. a unique point per cell), this heatmap could probably allow us to obtain cell center positions. Nevertheless, our model is still not perfect: final AP remains below 0.5, and for really crumpled DataMatrix, here are obtained predictions:

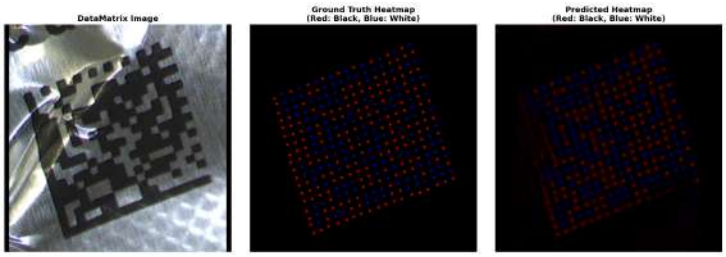

Figure 18: Training 4 crumpled DataMatrix predictions

3.3 Analysis of Environmental Debt Imposed by This Project

As mentioned previously, I developed a script allowing evaluation, at the end of each training, of environmental debt imposed by my model. This amounts in total, for all training runs I performed, to 4.64 kg of CO_2, which represents approximately the equivalent of a 30 km car trip. This data may seem relatively low, but must be contextualized. Indeed, it should not be forgotten that my training runs use pre-trained ResNet-18 weights, which were themselves obtained following training that probably emitted tens of tons of CO_2. On the other hand, it should be remembered that my project does not end with this report and could continue by deploying these training runs on Catholic University of Louvain clusters. This is

why I decided to take a broader interest in the environmental impact of digital at Catholic University of Louvain, as presented here : (°).

4 Conclusions and Future Research Directions

4.1 Summary of Achievements

This research internship at Catholic University of Louvain achieved a good portion of the objectives set in the initial specifications. Implementation of a deep learning method for predicting DataMatrix code cells in degraded conditions materialized through several contributions.

The developed architecture, based on ResNet with progressive upsampling decoder, demonstrates its ability to effectively process DataMatrix codes presenting complex geometric deformations. The hybrid approach combining offline transformations (curvature and crumpling) and online (geometric and photometric) significantly improves model robustness while limiting overfitting.

Implementation of a composite loss function combining MSE, Focal Loss and Contrast Loss constitutes an interesting methodological contribution. This combination effectively treats imbalance between background and cell pixels, producing results significantly superior to basic approaches with good convergence stability.

Provision of a structured GitHub repository responds to the engineering objective set initially. Modular scripts allow complete reproducibility of experiments and facilitate adaptation to other datasets.

4.2 Identified Limitations and Improvement Directions

Despite these advances, certain limitations persist. Heavily crumpled DataMatrix codes remain problematic, revealing insufficiencies of implemented crumpling transformations. This limitation stems from bilinear interpolation approximation, which does not capture complexity of real deformations. Moreover, material constraints limited experiments to ResNet-18 architectures.

Improvement of deformation transformations. Integration of 3D modeling tools like Blender would allow generating physically realistic deformations.

Architectural exploration. Evaluation of more sophisticated architectures (ResNet-50, EfficientNet, Vision Transformers) could reveal substantial gains. Results obtained with ResNet-18 suggest significant improvement potential.

Complete associative embedding. Transition from current Contrast Loss to a true associative embedding system represents a natural evolution allowing finer discrimination between black and white cells.

4.3 Overall Assessment

This internship allowed substantial deepening of skills in deep learning applied to computer vision. Theoretical mastery was accompanied by practical development in software engineering.

Identified perspectives offer concrete avenues for continuation of this work within Sarra Laksaci's thesis or by other teams interested in this problem.

5 Appendices

5.1 Addendum: Digital Carbon Footprint of Catholic University of Louvain

This analysis was made possible thanks to consultation with a fellow researcher from the laboratory, who indicated that all data concerning this subject was contained in Théo Gladsteen's study (2021) (-). Consequently, all my sources will refer to this paper.

5.1.1 Scale of Digital Emissions

UCLouvain's digital carbon footprint amounts to **5,446 tons of CO_2 equivalent** for the year 2019, representing nearly 8% of the university's global footprint. To contextualize this value, this is equivalent to traveling more than 760 times Earth's circumference by car.

This 8% proportion positions digital as the third emission sector after transportation and energy, making it a significant but not majority contributor to the university's overall carbon balance.

Distribution of emission sources

Detailed analysis reveals a surprising hierarchy of contributors:

- **Private student equipment**: 32.9% (1,793 tCO_2-eq)
- **Data transfer**: 27.7% (1,505 tCO_2-eq)
- **Office equipment**: 16.2% (881 tCO_2-eq)
- **Scientific data centers**: 5.9% (319 tCO_2-eq)
- **IT personnel**: 5.7% (309 tCO_2-eq)

The main sources of impact largely escape the university's direct control, with private equipment alone representing 36.4% of the total footprint.

5.1.2 Individual Emission Profiles

Impact per individual is distributed as follows:

- **Standard employee**: 235 $kgCO_2$-eq/year
- **Employee using scientific centers**: 1,538 $kgCO_2$-eq/year
- **Student**: 118 $kgCO_2$-eq/year

5.1.3 Characteristics of the Digital Sector

Predominance of embodied emissions

The digital sector presents a notable specificity: embodied emissions, i.e. CO_2 emissions generated during equipment manufacturing (raw material extraction, component production, assembly), largely dominate usage emissions for most digital equipment. This characteristic contrasts with other domains like automotive, where only 17% of a car's total emissions over 200,000 km come from its manufacture, the rest being related to its use (fuel consumption).

This particularity is explained by the fact that the relatively low-carbon Belgian electricity mix reinforces the dominance of embodied emissions.

Rapid growth of data transfer

The explosion of transferred data volume constitutes a worrying aspect: in 2019, UCLouvain exchanged a total of 22.81 petabytes of data – equivalent to approximately 5.7 million HD films or 22 million hours of video. In 2017, this volume was only 15 petabytes, representing a 50% increase in just two years.

5.1.4 Uncertainties and Probable Underestimation

Several elements suggest an **underestimation of results**:

Undervalued emission factors

Although the results presented above already seem considerable, they are probably still underestimated. Comparative analysis conducted on 100 laptops reveals significant gaps with official ADEME estimates (French Agency for Environment and Energy Management). For an 11 inch laptop, predicted carbon

impact is 29 % higher than ADEME reference values. This gap widens with equipment size: for an 18 inch laptop, the difference reaches 66 %. Official ADEME estimates therefore systematically undervalue the real environmental footprint of computer equipment.

Sensitivity analysis

In an unfavorable scenario combining all uncertainties, the footprint could reach 7,330 tCO_2-eq, a 35% increase, then representing 10.4% of UCLouvain's global footprint.

5.1.5 Reduction Potential

Evaluation of improvement scenarios reveals levers of variable efficiency:

Measure	**Reduction**
Extended lifespan (+1 year)	-7.1%
Replacing desktops with laptops	-8.8%
100% renewable electricity	-3.5%
Generalization of 13" screens	-0.7%
Total combination	**-21.1%**

Table 1: Reduction potential per measure

The most effective actions aim to limit the number of equipment, favor models with lower embodied footprint, or maximize their lifespan.

5.1.6 Impact of COVID-19 Crisis

Complementary 2020 analysis reveals contrasting impacts:

- Institutional investments: 6.6 tCO_2-eq with 35% uncertainty
- Private investments: 208.3 tCO_2-eq with 26% uncertainty
- Microsoft Teams usage: 327 tCO_2-eq with 66% uncertainty

UCLouvain was already relatively well equipped with digital technologies, which could explain the low investment needed for lockdown.

5.1.7 Conclusions and Implications

The digital sector constitutes a strategic issue for the 2035 carbon neutrality objective due to its rapid growth and predominance of private equipment.

Three major challenges emerge:

1. Main sources of impact escape the university's direct control
2. Rapid sector growth (+50% in 2 years)
3. Importance of embodied emissions difficult to optimize

Raising awareness in the university community about the impact of private equipment appears as a priority recommendation, achieving carbon neutrality requiring an approach combining institutional actions and individual behavioral changes.

5.2 Definitions

Dataset: Structured collection comprising DataMatrix code images and their associated annotation files.

Annotations: Document containing DataMatrix segmentation and characterization information, notably positioning coordinates and cell colors.

Keypoint: 2-dimensional vector indicating the vertical and horizontal position of a cell on a DataMatrix. These keypoints are recorded in the annotation file.

CNN Architecture: Structure specialized in image processing, composed of convolutional layers that apply filters to detect visual patterns (edges, textures, shapes), pooling layers to reduce dimensionality, and fully connected layers for final classification.

Overfitting: fundamental phenomenon in machine learning where a model learns excessively the specificities of the training dataset, to the detriment of its generalization capacity.

Epoch: designates a complete training cycle where the model has processed the entire training dataset once.

Batch: designates a subset of training examples processed simultaneously by the model during a learning iteration. In our case, this designates a subset of images loaded simultaneously by the model.

Training: automatic optimization process of neural network parameters so that it learns to perform a specific task from annotated data.

Hyperparameters: are configuration parameters of a machine learning algorithm that must be defined before training and are not automatically learned by the model.

Perlin Noise: pseudo-random noise function developed by Ken Perlin in 1983, widely used in computer graphics to generate natural textures and organic deformations. Unlike purely random white noise, Perlin noise produces coherent and continuous variations in space, with smoothing properties that avoid brutal discontinuities. This function generates values between -1 and 1 by combining several noise octaves at different frequencies, allowing fine control of generated deformation appearance. In the context of crumpling simulation, Perlin noise could have provided natural spatial displacements.

Learning Rate The learning rate η constitutes the fundamental hyperparameter that controls the amplitude of model weight updates during gradient descent. A learning rate too high can cause oscillations or optimization divergence, while a rate too low significantly slows convergence.

Weight Decay Weight decay constitutes a regularization technique that penalizes excessive parameter growth of the model by adding a penalty term to the loss function.

5.3 Library Descriptions and Python Implementation

Sklearn.cluster is the scikit-learn module dedicated to clustering (unsupervised grouping). It contains several algorithms to automatically identify groups in data:

- KMeans: partitions into k spherical clusters
- DBSCAN: finds variable density clusters with outlier detection
- AgglomerativeClustering: bottom-up hierarchical clustering
- SpectralClustering: uses graph theory for complex shapes
- GaussianMixture: models clusters as Gaussian distributions

Albumentations: Python library dedicated to data augmentation for computer vision. In this project, it allowed us to apply geometric and visual transformations to images in our dataset.

DataLoaders and data management PyTorch's `DataLoader` constitutes the standardized interface to feed the model with data during training and evaluation. This class encapsulates the custom dataset and automatically manages grouping into mini-batches, sample shuffling, and parallel data loading. In

implementation, two distinct DataLoaders are created: one for training with `shuffle=True` to randomize sample order at each epoch, and one for validation with `shuffle=False` to ensure deterministic evaluation.

5.4 Theoretical Elaboration of Implemented Methods

COCO annotation format COCO annotation files used in this project followed the standard JSON structure organized into several main sections. The *"info"* section contained general dataset metadata (description, version, contributor, creation date). The *"licenses"* section defined license information associated with data. The *"categories"* section specified detectable object classes, in our case a single *"datamatrix"* category. The *"images"* section grouped each image's metadata (unique image identifier, image file name, standardized dimensions of 512×512 pixels). The *"annotations"* section constituted the file core particularly exploiting the *keypoints* system: each annotation contained lists of triplets: (x, y, visibility) which precisely encoded spatial position with x and y and binary class with v (black/white) of each cell. This section also included each annotation's identifier, the image identifier corresponding to the annotation identifier, bounding boxes (*bbox*) defining rectangle coordinates delimiting the DataMatrix code on the image, as well as area (*area*) corresponding to the surface occupied by the code.

Theoretical elaboration of my curvature transformation This transformation applies a horizontal sinusoidal deformation to the image, creating a curvature or undulation effect. Mathematically, the transformation is defined by:

$$x_{source} = x_{dest} - \alpha \cdot \sin\left(\frac{2\pi y_{dest}}{H}\right) \cdot W \tag{2}$$

$$y_{source} = y_{dest} \tag{3}$$

where (x_{dest}, y_{dest}) represent destination coordinates in the transformed image, (x_{source}, y_{source}) corresponding coordinates in the source image, α the curvature factor, H the height and W the width of the image. This transformation preserves vertical coordinates while applying a sinusoidal horizontal shift whose amplitude varies periodically according to vertical position. Maximum shift amplitude is $\alpha \cdot W$, creating a deformation that oscillates between the left and right edges of the image. To maintain geometric consistency, keypoints undergo the same transformation, with a horizontal shift calculated by $\Delta x = \alpha \cdot \sin(2\pi y/H) \cdot W$, followed by clipping to ensure they remain within image boundaries.

Theoretical elaboration of my crumpling transformation The mathematical process of this transformation breaks down into several steps. First, n crumpling points are randomly generated in the image's central region (between 20% and 80% of dimensions) with a minimum distance between them to avoid overlaps. For each crumpling point located at (w_x, w_y) with a random radius R, the transformation calculates spatial influence by a Gaussian function:

$$\text{influence}(x, y) = \exp\left(-\frac{d^2}{2(1.5R)^2}\right) \tag{4}$$

where $d = \sqrt{(x - w_x)^2 + (y - w_y)^2}$ is the Euclidean distance to the crumpling point. The displacement then combines radial and tangential components with sinusoidal oscillations:

$$\Delta x = \alpha \cdot \text{influence} \cdot \left[\cos(\theta + d/R)\sin(2d/R) + 0.2\sin(2\theta)e^{-d/(0.8R)}\right] \tag{5}$$

$$\Delta y = \alpha \cdot \text{influence} \cdot \left[\sin(\theta + d/R)\sin(2d/R) + 0.2\cos(2\theta)e^{-d/(0.8R)}\right] \tag{6}$$

where $\alpha = I \cdot \text{wrinkle\_intensity}$, with I a random intensity implemented for each crumpling point, while wrinkle_intensity is the transformation's global intensity independent of the considered point, allowing manual adjustment to maintain consistency between annotations and transformed images. Moreover, θ is a random angle for each crumpling point which notably allows the transformation not to be symmetric around a crumpling point.

The term $\cos(\theta + d/R)\sin(2d/R)$ allows me to reproduce local curvature transformation, while the term $0.2\sin(2\theta)e^{-d/(0.8R)}$ allows me to reproduce local impact at the material level that generated this crumpling.

To apply this transformation to keypoints, bilinear interpolation of the displacement field is necessary because keypoints generally do not correspond exactly to integer pixel coordinates. Given a keypoint with coordinates (x_k, y_k), the transformation first calculates adjacent integer indices: $x_{int} = \lfloor x_k \rfloor$ and $y_{int} = \lfloor y_k \rfloor$, as well as corresponding fractions: $x_{frac} = x_k - x_{int}$ and $y_{frac} = y_k - y_{int}$. Bilinear interpolation then uses the four displacement values at corners of the encompassing pixel:

$$\Delta x_k = \Delta x_{00}(1 - x_{frac})(1 - y_{frac}) + \Delta x_{01} x_{frac}(1 - y_{frac}) \tag{7}$$

$$+ \Delta x_{10}(1 - x_{frac}) y_{frac} + \Delta x_{11} x_{frac} y_{frac} \tag{8}$$

where Δx_{ij} represents horizontal displacement at point $(x_{int} + i, y_{int} + j)$. The same formula applies for Δy_{ij}. I implemented this keypoint transformation according to the principle that main contribution comes from the nearest integer coordinate point. This interpolation moreover ensures continuous and differentiable keypoint transformation. It is this interpolation technique that led me to choose this approach rather than Perlin noise.

Theoretical elaboration of my decoder I propose here to illustrate how, from a heatmap provided by our backbone (ResNet), we can construct another that will have a dimension of 512×512 and a single channel if we are in unified mode, or two channels otherwise. This construction involves implementing new weights, but it is all weights of our backbone and our decoder that will be adjusted during our training. This decoder design is inspired by existing research works, while its implementation was done relying mainly on Python tutorials.

And finally

We start by applying this command at the backbone output:

```
nn.ConvTranspose2d(C_{in}, C_{out}, kernel_size=4, stride=2, padding=1)
```

where:

- C_{in} = number of input channels
- C_{out} = number of output channels
- H_{in}, W_{in} = input spatial dimensions

This command executes several steps.

Let our input tensor of shape $(N, C_{in}, H_{in}, W_{in})$ where:

- N = batch size

I set kernel size to 4 here, as it is often recommended with transposed convolution to set it to twice the stride value to limit artifact problems.

Step 1: Upsampling by zero insertion

For an input $\mathbf{X} \in \mathbb{R}^{H_{in} \times W_{in}}$, and for stride=2 we create:

$$\mathbf{X}_{upsampled} \in \mathbb{R}^{(2H_{in}-1)\times(2W_{in}-1)\times(C_{in})} \tag{9}$$

Example for a 2×2 matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow \begin{bmatrix} a & 0 & b \\ 0 & 0 & 0 \\ c & 0 & d \end{bmatrix} \tag{10}$$

Step 2: Convolution with 4×4 kernel (kernel_size = 4)

The kernel $\mathbf{K} \in \mathbb{R}^{4\times4\times C_{in}\times C_{out}}$ is applied:

$$\mathbf{K}[k,c] = \begin{bmatrix} w_{1,1,k,c} & w_{1,2,k,c} & w_{1,3,k,c} & w_{1,4,k,c} \\ w_{2,1,k,c} & w_{2,2,k,c} & w_{2,3,k,c} & w_{2,4,k,c} \\ w_{3,1,k,c} & w_{3,2,k,c} & w_{3,3,k,c} & w_{3,4,k,c} \\ w_{4,1,k,c} & w_{4,2,k,c} & w_{4,3,k,c} & w_{4,4,k,c} \end{bmatrix} \tag{11}$$

where c represents the output channel with $c \in \{0, 1, \ldots, C_{\text{out}} - 1\}$ and where k represents the input channel with $k \in \{0, 1, \ldots, C_{\text{in}} - 1\}$.

Transposed convolution produces:

$$(\mathbf{X}_{\text{upsampled}} \star \mathbf{K})_{i,j,c} = \sum_{k=0}^{C_{\text{in}}-1} \sum_{m=0}^{3} \sum_{n=0}^{3} \mathbf{X}_{\text{upsampled}}[i+m, j+n, k] \cdot \mathbf{K}[m,n,k,c] \tag{12}$$

where c represents the output channel with $c \in \{0, 1, \ldots, C_{\text{out}} - 1\}$.

Step 3: Padding application

padding=1 removes 1 pixel from each edge:

$$\mathbf{Y}_{final} = \mathbf{Y}_{conv}[1 : H_{conv} - 1, 1 : W_{conv} - 1] \tag{13}$$

Finally at end of process we obtain:

$$H_{out} = (H_{in} - 1) \times 2 - 2 \times 1 + 4 = H_{in}) = 2 \times H_{in} \tag{14}$$

Similarly:

$$W_{out} = 2 \times W_{in} \tag{15}$$

And moreover:

$$C_{out} = 1/2 \times C_{in} \tag{16}$$

Thus, by repeating this process several times, we finally obtain a heatmap of dimensions 512×512 with 16 channels.

On the other hand, between each iteration of this process, I also apply three essential operations:

BatchNorm2d: For each deconvolution output channel, this layer calculates the mean and standard deviation of all pixels of this channel over the entire batch. It then normalizes each pixel by subtracting this mean and dividing by standard deviation. This normalization prevents values from exploding or vanishing, stabilizes training and allows using higher learning rates.

ReLU: This activation function goes through all channel pixels and applies a simple rule: if a pixel has a negative value, it sets it to zero; otherwise, it keeps its value. This operation introduces non-linearity essential to the network – without it, regardless of network depth, it would remain fundamentally linear.

Dropout2d(0.1): During training, this regularization randomly selects 10% of entire channels and completely sets them to zero. During evaluation, it makes no modification. This technique forces the network not to depend excessively on certain specific channels, which improves its generalization capacity and reduces overfitting risk.

Finally, we apply a final convolution allowing to reduce the number of channels (1 in unified mode, 2 otherwise), then we apply the sigmoid function to constrain values between 0 and 1, which is necessary for loss function implementation.

Elaboration of my PCK (Percentage of Correct Keypoints) implementation: My implementation follows a multi-step process: first, keypoints are extracted from heatmaps by detecting local maxima above a threshold, then localization accuracy is calculated via Euclidean distance analysis. Mathematically, let $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ be the set of predicted keypoints and $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ the set of ground truth keypoints. For each predicted keypoint p_i, minimum distance to ground truths is calculated as:

$$d_i = \min_{j \in \{1,\ldots,m\}} \|p_i - t_j\|_2$$

where $\|\cdot\|_2$ denotes Euclidean norm. A predicted keypoint is considered correct if $d_i < \tau$, where τ represents the tolerance threshold (which I set to 20 pixels by default). Final PCK score is then expressed as:

$$\text{PCK} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(d_i < \tau)$$

where $\mathbf{1}(\cdot)$ is the indicator function.

Elaboration of my training carbon footprint calculation

My estimation of training environmental impact relies on an energy calculation model based on continuous GPU consumption measurement. Instantaneous power $P(t)$ (in Watts) is sampled every 5 seconds via `nvidia-smi` interface, generating a time series $\{P_i\}_{i=1}^{N}$ over total training duration T (in hours). Average power is calculated as $\bar{P} = \frac{1}{N}\sum_{i=1}^{N} P_i$. Total energy consumed is then expressed by $E = \bar{P} \cdot T$ (in Wh), converted to kilowatt-hours: $E_{kWh} = \frac{E}{1000}$. Carbon footprint is finally estimated via $C = E_{kWh} \cdot f_{CO_2}$, where $f_{CO_2} = 0.11$ kg CO/kWh represents the Belgian energy mix emission factor. This approach allows quantitative evaluation of environmental impact of different training configurations, facilitating comparison between architectures and optimization strategies.

AdamW elaboration

The AdamW optimizer (Adaptive Moment Estimation with Weight decay) combines advantages of Adam's moment adaptation (another optimizer) with decoupled L2 regularization. Decoupled L2 regularization means that the weight penalty ($\lambda\theta_t$) is applied directly to parameters independently of adaptive moments, unlike classic Adam where it passes through gradient calculation and interacts with moment estimations. The algorithm maintains two exponentially decaying estimations: first moment $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$ which captures gradient moving average, and second moment $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$ which estimates gradient variance, where $g_t = \nabla_\theta \mathcal{L}(\theta_t)$ represents gradient at iteration t. These moments are then corrected for initialization bias: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$ and $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$. Parameter update is performed according to:

$$\theta_{t+1} = \theta_t - \eta\left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda\theta_t\right)$$

where θ represents all convolutional neural network parameters, η is the learning rate controlling global update amplitude, $\epsilon = 10^{-8}$ avoids division by zero and λ is the weight decay coefficient. Hyperparameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ respectively control decay of gradient and its variance moving averages, allowing stable convergence even with noisy gradients.

5.5 Sources and Research Papers

Title: Top 200 Universities in Europe
Author: uniRank
URL: `https://www.unirank.org/europe/top-200/`

Title: U-Net: Convolutional Networks for Biomedical Image Segmentation
Authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox
Date: 2015
URL: `https://arxiv.org/abs/1505.04597`
Description: This paper introduces the U-Net architecture, a convolutional neural network for biomedical image segmentation.

Title: Deep Residual Learning for Image Recognition
Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
Date: 2015 (submitted), 2016 (published)
URL: `https://arxiv.org/abs/1512.03385`
Description: This foundational paper introduces Residual Networks (ResNets), revolutionizing very deep neural network training through residual connections (skip connections).

Title: Associative Embedding for Game-Agnostic Team Discrimination
Authors: Maxime Istasse, Julien Moreau, Christophe De Vleeschouwer
Date: 2019 (submitted), 2019 (published)
URL: `https://arxiv.org/abs/1907.01058`
Description: This paper proposes a method based on a convolutional neural network (CNN) to learn pixel-wise descriptors as embedding vectors, similar for pixels representing players from the same team and dissimilar for pixels corresponding to distinct teams.

Title: The Digital Carbon Footprint of UCLouvain
Authors: T. Gladsteen
Date: 2021
Description: This study analyzes the digital carbon footprint of Catholic University of Louvain, evaluating the environmental impact of the institution's digital activities.