École Polytechnique Fédérale de Lausanne

Reconstructing lensless image with ML models and deploying them onto embedded systems

by Jonathan Reymond

Master Thesis

Approved by the Examining Committee:

Dr. sc. Paolo Prandoni
Thesis Advisor

Prof. Nicholas Antipa
External Expert

Eric Bezzam
Thesis Supervisor

EPFL IC IINFCOM LCAV
BC 341 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 30, 2023

Dedicated to the ones who have prayed for me.

Acknowledgments

I would like to thanks everybody who have helped me to review this thesis, namely Hanna Schwéry, Olin Bourquin and Noémie Reymond. I would also like to thanks to all my family for their constant support within all these years. I am greatful for the LCAV lab for providing me a really pleasant working environment. And finally, I would like to thanks Eric Bezzam for all the help is has given to me these past months, by reviewing my work and giving me good advices.

Lausanne, June 30, 2023 Jonathan Reymond

Abstract

Lensless imaging provides a large panel of benefits : cost, size, weight, etc., that are crucial for wearable application, IoT or medical devices. Such setups require the design of reconstruction algorithms to recover the image from the captured measurements. Most of the current SoTA reconstruction models use deep learning, but the results provided are hardly reproducible and mostly not meant to be deployed into embedded systems.

In this work, we implement the work of Monakhova et al.[16] that use uniquely deep learning, and the work of by Khan et al.[11]. We then present a way to transform these models to be deployable using TensorFlow Lite, and evaluate the benefits of model optimization techniques such as quantization-aware training(QAT), weight pruning, or weight clustering.

Contents

Chapter 1

Introduction

In recent years, technology has been improving at incredible pace. In many fields this trend can be observed: wearables, IoT, AI, medical, etc. With this also comes the need of miniaturizing hardware components and lowering the cost to be affordable by most people. The imaging field is not an exception, where we want to reduce the size of the system as much as possible. However, one major problem emerges with our current cameras: the lens. It cannot be indefinitely miniaturized due to physical/optical constraints, is expensive to produce, and is heavy compared to the whole system.

These considerations have pushed the research community to study lensless cameras extensively in the past decade. Lensless imaging involves the use of algorithms to perform image formation in the digital post-processing, rather than directly on the sensor. Although a lot of work has been made to explore the best ways to reconstruct these images, only a low amount of effort has been deployed to make the results reproducible, and thus making the comparison between various works very tedious. Also, most reconstruction techniques have a very long running time, and almost all of them are currently not suitable to be deployed into an embedded system, taking into account computational and memory constraints.

Concerning the running time of the reconstruction algorithms, some comparisons have been made between classical methods (ADMM, Fista, ...) and deep learning methods, but the analyses proposed are only comparing Graphics Processing Unit (GPU) running time, which is a setup that we rarely encounter in practical edge computing scenarios. To our knowledge, no work has been realized to try to take any reconstruction algorithm and deploy it to a constrained device.

To solve these two issues, we reimplement two approaches presented by Monakhova et al.[16](the U-Net model), and by Khan et al.[11] and try to reproduce the results stated in their paper. Both are purely deep learning based models. We have implemented a framework to be able to deploy these models into any embedded system, and then we evaluate to see what are the costs in terms of performance and accuracy. One of the motivations is that nowadays, reconstruction

solutions using machine learning are the ones producing the most realistic and/or accurate results, and thus are the ones that will probably be considered for deployment for commercial use in the future. Concerning the deployment phase, we have tested various techniques, such as pruning, clustering, quantization aware training (QAT) to shrink the models so that the resulting model could obtain better inference-time and satisfy the resource constraints of the targeted device.

1.1 Related work

For ML tasks having as input and as output of the model an image, as we have, there is image segmentation, where multiple papers used U-Nets[18][27][7]. Another tasks related are image denoising[28][24], or image super-resolution[25][22]. Various models have been successfully deployed into embedded systems, such as DeepLab[8] for image segmentation, or ESRGAN[22] for image super-resolution. No models for lensless reconstruction were made for deployment to our knowledge.

In the context of lensless imaging, Bezzam et al.[4] built a privacy-preserving system that captures the scene with a lensless camera and a programmable mask to encode the image, all within an embedded system.

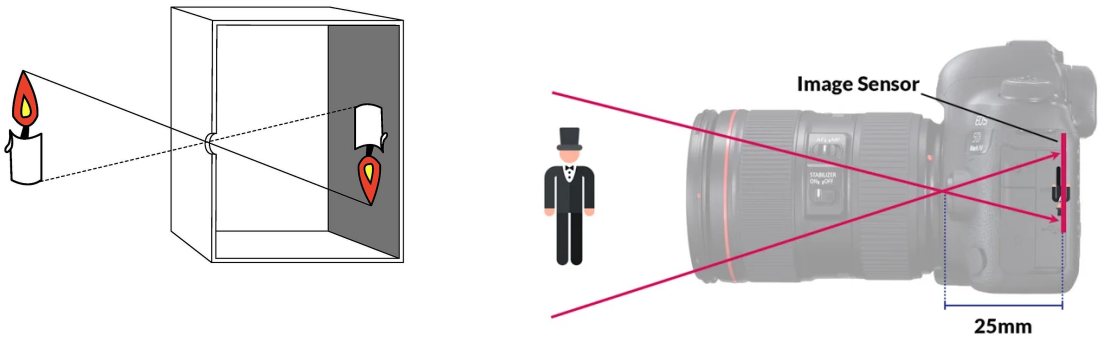
Chapter 2

Lensless imaging

2.1 Context

Before explaining the choice of using lensless cameras, let's depict the context. At the beginning of the history of cameras, it was basically only a box with, on one side, a pinhole, and on the other side, an analog sensor, or a photographic film, to capture the image on the other side. As showed in the Fig 2.1., the scene is projected directly at the back of the box, only reversed, but already in a representation similar to the human eye. The drawback of this technique is that it requires a long exposure time: since only a small amount of light passes through the pinhole, one needs to place the camera for a long time to collect enough light to see the image clearly on the photographic film. As a countermeasure, we have added to the system the lens with the idea of collecting a large amount of light and thus reducing the exposure time. Naturally, other advantages of using a lens appear, such as zooming, or increasing the field of view, but they are outside the scope of this master's thesis. There are, however, a lot of other problems with lenses, in particular their cost, representing more than 90% of the cost of a whole camera. Furthermore, due to the lens, we can not reduce the size or the weight of the system as far as we wanted.

At the time of analog cameras, we needed to use lenses, due to the type of light sensors: analog. We were forced to directly produce an image that is the exact reproduction of the scene from the human point of view. But with the apparition of digital sensors, this constraint is removed. Indeed, one could think of a system that gathers light information from the scene, stores an image of it that does not look like anything from the human perception, but contains all the information for a computer to be able to reproduce the scene in a second step such as our eyes would see it. With this idea in mind, the research community has begun to study lensless cameras in recent years. When we remove the lens from the equation, what do we have ? A pinhole camera that suffers from not collecting enough light in a short period of time. But what if instead of only one small hole, one adds multiple holes in the box ? One would be able to capture way more light, and at the end,



(a) pinhole camera (b) lens camera

Figure 2.1: Basic camera principle

what would be captured by the sensor would be just a multiple images of the scene mixed together. It is clear that the captured image would be very blurry for us, but now since it is digitally stored, one could develop a method to process it and demultiplex this image within the computer, and showing only in the end the result of the process. As explained previously, using lensless-based camera yields to multiple benefits: including the weight and the cost of the system, but also the scalability of constructing such cameras due to their simplicity. Also, for some applications, for instance microscopy, to increase the resolution, one has to reduce the field of view, which is not the case with lensless cameras where the field of view only depends on the sensor size. There are numerous fields of application where such cameras could be beneficial, such as 3D imaging, IoT devices, wearable systems, microscopic imaging, etc...

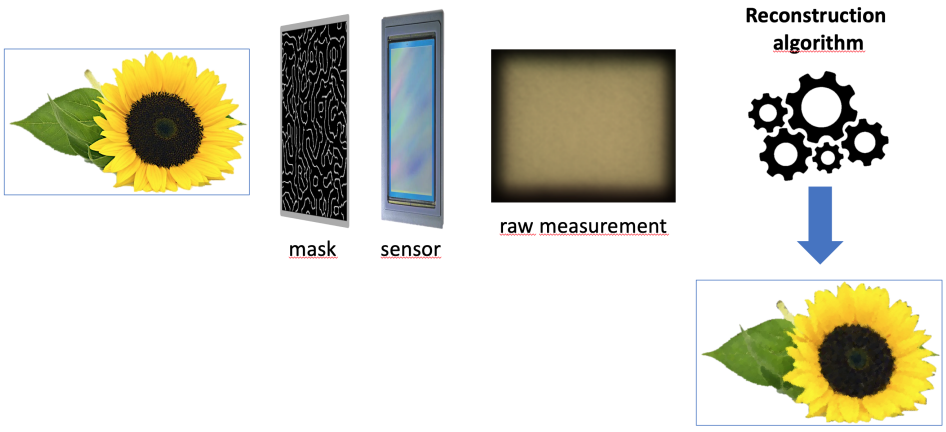


Figure 2.2: Lensless camera process

Naturally, the crucial point of such cameras is their efficiency to reconstruct the scene from the stored measurements is. Does it give a good resolution in the end? Also, one also needs to care about the complexity of the method, does it take a sufficiently short amount of time to be usable with the computational resources we have at our disposal for the application we are interested in? All these questions have yielded multiple designs of lensless cameras. For example, instead of adding multiple holes in the frontend of the box, one could replace it with a glass diffusing the light instead. All these types of frontend apertures are called masks, and various designs have been made to facilitate the reconstruction of the images. Also, various methods/algorithms were developed for reconstruction, as we explain a bit later.

2.2 Modeling

2.2.1 Linear inverse problem

There are multiple ways to describe the problem of reconstructing the image from the measured data. A common approach is to view it as an inverse problem: we know the process used to generate the measured data, which is called the forward model, and we want to reverse it. Here in this paper, we will simplify the problem by saying that the scene is only a 2D image, where each location of the image depicts the light's intensity. Then we go a step further by stating that the relationship linking the measurement and the original scene is linear, which is the case for the system designs we explore later. It yields the following relation:

$$y = Hx,$$

where y is the measured image, x the original image, and H the forward operator, or forward model, which is, simply a linear operator and is the part that depends on the camera design. Ideally, one would try to find the inverse of H to get $H^{-1}y = x$, but this is infeasible since H^{-1} is undetermined, and doing so could also amplify noise. Furthermore, taking the example in [6], if the original image and the measured data have a shape of the order of one megapixel, then the number of elements of H would be close to 10^{12} elements, so at least one Terabyte of flash memory to store it, and computing the matrix product of $H^{-1}y$ would require also 10^{12} multiplications, so $O(N^2)$ in both cases where N is the number of pixels, so clearly too complex to be considered.
We know that H depends on the choice of the aperture/mask we use. Do there exist masks that simplify this relation? Thankfully, the answer is yes.

2.2.2 Mask design

Before detailing the theoretical properties of each mask, we will first show what type of mask we have studied in the context of this project. The simplest one is the first we have considered, where the idea is to have multiple holes at the surface. Called the **amplitude mask**, it lets the light pass or not, and so modulates the received amplitude by 0 or 1. Then we have the **phase modulation-based mask** family, where we have a transparent material, a thin glass or plastic, with a different thickness on each location, slowing more or less the light and thus involving a phase modulation in the frequency domain. We have the **phase grating** where the modulation is either 0 or π, the **phase mask** where the modulation takes discrete values between 0 and π, and finally the **diffuser** one where the modulation is continuous. See Fig. 2.3.

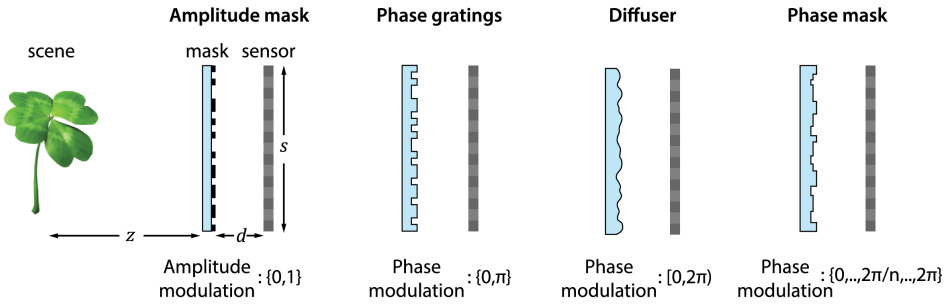


Figure 2.3: Different types of phase masks

2.2.3 Mask properties

Separable property

In this case, if the pattern in the mask is made by the cross-product of two vectors, then the relation in 2.2.1 can be simplified as follows :

$$\mathbf{Y} = \Phi_L \mathbf{X} \Phi_R^T,$$

where Y is the 2D measured data, X the 2D original image, Φ_L the linear operator modifying the rows of the image and Φ_R^T the columns. To illustrate the memory and computational gains, if we take the same settings as 2.2.1, we need to store only two $10^3 \times 10^3$ matrices, and doing the multiplication in parallel yields to $O(N)$. Amplitude, phase-gratings and phase masks could be designed to fall into this category. This concept was first developed by DeWeert et al.[10], and one another example of a lensless camera having a mask holding this property is the FlatCam[3].

Convolutional model

PSF The masks that we study can all be described using their Point-Spread Function (PSF). The PSF of a mask is the measurement made with a constant, fixed single source point of light, see Fig. 2.4.

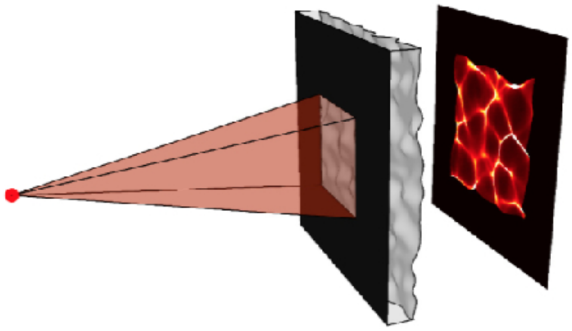


Figure 2.4: Point-spread-function (PSF) measurement

The idea behind this measurement is simply to evaluate the impulse response of the overall system, i.e. how it alters the light that passes through.

Convolutional property To be able to use the conventional property, we have to assume that the PSF is shift-invariant, that is, when we change the position of the point-source in Fig. 2.4, the resulted image is the same as the original one, but transposed/shifted. This assumption is satisfied under some conditions : the system has a narrow field of view, the scene we want to picture is sufficiently far away and the distance between the mask and the sensor is not too big (see [6]). The convolutional property approximates 2.2.1 to the following relation :

$$Y = X * h$$

Where Y and X are the 2D images representing the measurement and the original scene respectively, and h being the measured PSF of the system. When regarding the memory consumption, one needs to store only h, who has the same size as Y, so only $O(N)$ where N is the number of pixels. And computing the convolution in the Fourier domain instead results in a $O(NlogN)$ complexity, so undoubtedly manageable.

Not all applications are suited for this use, such as microscopy where the scene is not sufficiently far away from the camera to hold the shift-invariant property. Kuo et al.[14] described a way to handle this case by adapting slowly h to do local convolutions. Most of the diffuser mask-based cameras attempt to satisfy this property, such as PhlatCam[5] or DiffuserCam[13]. As a side note, some lensless cameras can satisfy both properties under certain conditions, such as the FlatCam

camera, while keeping in mind that the separable case is the strongest property between both.

2.2.4 Optimization techniques

There exist numerous ways to solve linear inverse problems. We will only name a few here, from the more standard to the more AI-based.

Classical methods

Without dtailing too much, the classical way of solving inverse problems is to define an objective function we want to minimize that measures or close is our reconstruction close to the target. In our setup, we have the following objective function :

$$\hat{x} = \underset{\boldsymbol{x_p} \in \mathbb{R}^N}{\arg\min}\, F(\boldsymbol{y}, \boldsymbol{H x_p}) + \lambda \mathscr{R}(\boldsymbol{x_p})$$

H being the forward operator, y the measured data, x_p the tunable object that at the end of the minimization gives our proposed reconstructed image, F the function that measures the data-fidelity between the measured data and the proposed reconstruction passed through the forward model, $\mathscr{R}$ the regularization function to favour some given set of possible solutions and λ a value controlling how much we want to regularize our solution. The standard data-fidelity function used is the mean-square error, that is $\frac{1}{2}\|\mathbf{y} - \mathbf{H}\mathbf{x_p}\|_2^2$. Multiple types of regularizations are employed in the context of images, such as the L_0 (counting the number of non-zero elements in x_p) and L_1 norm ($\|\mathbf{x}\|_1$), which enforce the solution to be sparse and therefore yield to more sharp images, or the Tikhonov and the TV norm to control the smoothness of the image, etc. Depending on the data-fidelity and regularization functions employed, different optimization algorithms can be applied, such as APGD, Fista, or ADMM. As an example, in one of the papers we study[16], the optimization problem is formulated as :

$$\hat{\mathbf{X}} = \underset{\mathbf{X_p} \geq 0}{\arg\min} \frac{1}{2}\|\mathbf{Y} - \mathbf{h} * \mathbf{X_p}\|_2^2 + \lambda \|\Psi \mathbf{X_p}\|_1$$

Where Ψ is a transform operator, such as the one to get the TV norm. Practically, all these algorithms are iterative and need a certain number of steps to converge to the minimum. Given a sufficiently small step size and a convex objective function, these algorithms have the guarantee to converge to the global minimum.

Deep learning

A radically different approach is to completely ignore the lensless setup and the inverse problem formulation, and to try to reconstruct images given a dataset and a machine learning model. Practically, the dataset has pairs of measurements and original images $\{(Y_i, X_i)\}_{i=1}^N$, it feeds the measurement Y_i through the ML model f_w, where w are the learnable parameters/weights of the model. Then we compute a given loss that measures how close is the output of our model with respect to the target image X_i. Then we update the model with gradient-based optimizer via backpropagation so that in the end it renders sufficiently good images. In contrary to the classical methods, the loss is highly non-convex and therefore gives no guarantees that the final solution is a global minimum. More specifically, if we take again the mean-square error as the loss, the function that we want to minimize in this context would be :

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \|X_i - f_w(Y_i)\|_2^2$$

and since f_w itself in the deep learning context where we pass the input through multiple non-linear layers, it results that the whole function is highly non-convex. Nevertheless, using deep learning in this area leads to more realistic results, often at the cost of data-fidelity . And having at our disposal a GPU, the time it takes to run one inference, i.e. passing one measurement through the model to get the reconstruction, is a few order of magnitude smaller than the classical methods (see [16]).

Unrolled optimization

Even though classical algorithms give convergence guarantees, the time the algorithm takes to converge relies directly upon the parameters controlling the step size, and the appearance of the resulted image depends highly on the chosen regularization and its factor λ. All these parameters must be fine-tuned to suit the given instance we want to reconstruct. But what if at each step we could choose the best hyperparameters for our specific problem if we know how the results should look like? For instance, having a large step size at the beginning, and smaller ones at the end of the process, or having at our disposal the regularization that suits perfectly the data we want to reconstruct. We would first decrease significantly the number of steps needed to converge, but also produce more realistic results due to this new regularization term. And that is what unrolled optimization tries to do. The basic concept of this method is to combine machine learning and the classical approach by building first a neural network with a fixed amount of layer where each layer is the implementation of one step of a given classical algorithm like Fista or ADMM, where the step size and also possibly the regularization is learnable, that is can be adapted. Naturally, we need to have at our disposal a dataset of pairs (X_i, Y_i) to train this model for our task so that the model learns how to produce a realistic X' given at input any Y'. On top of that, Monakhova et al.[16] has also suggested to add another model after the unrolled ML model that they call the "learned denoiser".

This model acts as a perceptual enhancer, so that the output looks closer to the human perception of nature.

Chapter 3

Deep learning models

The methods we have implemented were the ones presented in the papers [16], and [11]. The goal, as written previously, is first making their works reproducible, usable for other datasets and problem instances. Secondly, prepare them to be deployed into embedded systems, and to evaluate properly the practical inference time and precision.

3.1 Monakhova et al. approach

The paper from Monakhova et al.[16] presents multiple models. We study here only the fully-based deep learning model, namely the U-Net model. But foremost, let's consider the lensless camera setup used.

3.1.1 Lensless camera setup

The lensless camera employed is the DiffuserCam presented in [13]. As its name suggests, it is a lensless camera with a diffuser mask, and it has been set up to satisfy the convolutional property 2.2.3, i.e. at a sufficiently far distance so that a single PSF can be used for reconstruction. As a side note, even though for the U-net architecture described below the PSF nor the convolution property are used, they are used when we evaluate the FlatNet model [11] over this dataset. To generate the dataset of measurements Y and original/ground truth images X, the images were displayed by a screen and at one side the lensless system measures Y and a lensed camera at the other side captures X as showed in Fig. 3.1.

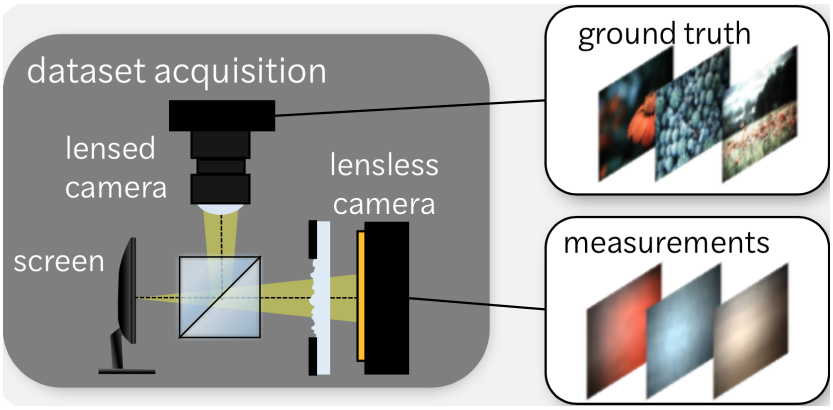


Figure 3.1: DiffuserCam dataset acquisition, from [16]

.

3.1.2 Model architecture

The U-net architecture presented first in 2015 by Ronneberger et al.[18] has been widely used on various image tasks, such as image segmentation, denoising , or even super-resolution. The principle relies on the autoencoder concept: we have the first part, the encoder that takes as input the image and reduces its dimension to a much smaller space that we call the latent space. The objective is to extract all the relevant information from the image in a very compressed way. This encoder part is typically a convolutional neural network (CNN) based model. It consists of multiple stages, where each stage contains multiple stacked convolutional layers followed by an activation function such as ReLU and a max-pooling layer. After each stage, the image height and width are reduced by a factor of two, while the number of channels/features are increased, leading at the end to a vector containing the features of the image. The second part of the U-Net is what we call the decoder. The decoder tries to reconstruct the image from the feature vector of the input. In our case, where the input is the measurement Y, the decoder tries to reconstruct an image close to the ground truth X from the feature vectors of Y generated by the encoder. The decoder architecture has also multiple stages, the same number as the encoder, with stacked convolutions layers and an activation, but instead of a pooling layer, it has an upsampling or a transposed convolution layer to expand again the height and the width of the signal to get at the end an output with the same shape as the target X. The manner in which the features are extracted by the encoder comes at a cost, namely, we lose the localization of these features within the image. Therefore, we add skip connections at the end of each encoder stages and before the decoder stages, where the temporal representation of the image at the given stage is concatenated to the temporal reconstruction of the decoder at the same stage, see Fig. 3.2. The number of stages, the model depth, as well as the number of filters used for each convolution define the representation power (or capacity) of the model.

In the context of our paper, the chosen U-Net has 5 stages, where the number of stacked convolutional layers is 2 for the encoder part, and 3 for the decoder part. For each stage, we define

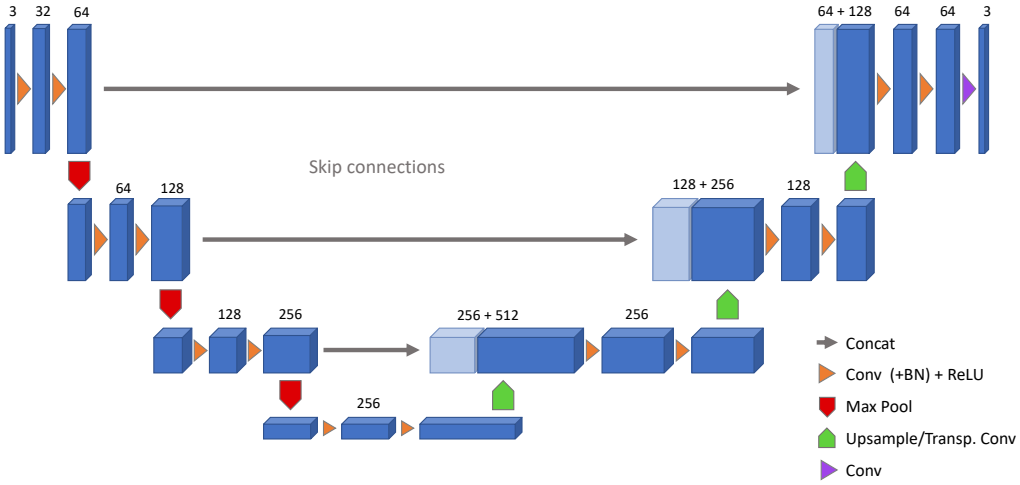


Figure 3.2: U-Net: basic architecture

.

an increasing number of filters : [24, 64, 128, 256, 512]. The kernel size is fixed at 3×3. To increase the height and the width, the upsampling method was used. We add two convolutions at the end of the model, the first one with 12 filters, and the last one with a kernel size of 1 and 3 filters to get 3 channels for RGB images. We designate in the following chapters this specific architecture as U-Net$_{\text{waller}}$.
Multiple more complex variants of the U-Net were elaborated since then, and some of them were incorporated in our implementation.

3.1.3 Loss functions

To evaluate the reconstruction made by the ML model against the ground truth image, one has to properly define the metrics and losses. In fact, we need a linear combination of different losses to capture all the desired properties at the end. The first part of the loss is the mean squared error (MSE) that enforces the solution to be close, in a pixel-wise sense, to the ground truth. Once we look more closely at the MSE, it gives smoother results: since the error is computed pixelwise, the best way for the algorithm to reduce the loss is to average the colour intensity with the neighbouring pixels. Furthermore, MSE, also due to this pixelwise behaviour, is unable to quantify the overall structure of the image. So, only this component is not sufficient to produce results that are satisfying and acceptable for the human perception.

To capture this feature, we introduce the Learned Perceptual Image Patch Similarity (LPIPS)[26].

The central concept is to pass the ground truth and the reconstructed output to a machine learning model that was trained over an external dataset of real images for another task, like classification. Then we take the output of different layers within this ML model, compute the MSE between the output of the reconstruction and the ground truth, and average the results, weighting each term by a certain factor. The main motivation behind this process is that the output of the image at a given layer is a compressed representation that is no more pixelwise dependent, but seek to extract the useful features of an image for the given classification task, e.g. edges or shapes within the image. For CNNs, it is commonly accepted that early layers capture more general features of the image, and as we go deeper, it becomes more specific for the given task [23]. So by computing the difference between the respective two representations at a given layer, we are effectively evaluating how close are their features. And since the ground truth features are perceptually realistic, the LPIPS enforces our model to enhance the realism of the output.

There exists therefore multiple type of LPIPS losses depending on the model chosen. The two popular choices are AlexNet[12], and a more recent CNN, VGG-16[20]. Both were trained on the ImageNet dataset[9]. ImageNet is a very large dataset having 3.2 million samples annotated with more than 5000 different labels. So any model trained over it has to extract extremely different and general features to be able to differentiate between all these classes, which is exactly what we want for our problem. As is the trend in machine learning, newer models quickly arise that outperform previous models. However, these newer ones are often more complex, making them very unpractical since we have to pass the samples through the model during training and backpropagate, and also store this model to the RAM of our system. We have decided to employ the VGG-16 LPIPS version as it has a better accuracy than the AlexNet for the ImageNet classification task.

3.2 FlatNet

A more developed, fully-based deep learning approach was proposed in [11]. A key contribution is to add a custom machine learning layer, that explicitly inverts the one-to-many mapping of the lensless camera system in Fig.2.2.1. In their paper, they consider two cases: arbitrary masks and the special case where we have a separable mask.

3.2.1 Separable case

The inversion operation we have to reverse is $\mathbf{Y} = \Phi_L \mathbf{X} \Phi_R^T$ as stated previously in section 2.2.3, where Φ_R and Φ_L are linear operators in matrix form, i.e. express X as a function of Y. It yields the following expression :

$$X_{interm} = f(W_1 Y W_2),$$

where W_1 and W_2 are representing the adjoint of Φ_L and Φ_R^T respectively, and f a non-linear function, in our case the leaky ReLU. Practically, W_1 and W_2 are learnable weights initialized with the values of their corresponding adjoint so that they can be better learnt to invert the lensless camera system, and f is here to help by allowing the inversion process to be also non-linear.

To evaluate this case, they have generated a dataset with the FlatCam setup. We refer to the induced ML model as FlatNet.

3.2.2 Non-separable/convolutional case

Here the system is modelled by $Y = X * h$ (see 2.2.3). The inversion is best done in the Fourier domain for computational efficiencyL

$$\mathscr{F}(Y) = \mathscr{F}(X) \odot H,$$

where H is the Fourier transform of the PSF h. The inversion is done in the Fourier domain and brought back to the spatial domain:

$$X_{\text{interm}} = \mathscr{F}^{-1}(\mathscr{F}(W) \odot \mathscr{F}(Y)).$$

For the initialization of W, we set it to $\mathscr{F}^{-1}(\frac{H^*}{K+|H|^2})$, H being the Fourier transform of the PSF and K a calibration parameter to avoid noise amplification. As you can observe, there is no non-linear function like the separable case. Experimentally, the authors of [11] have found that it does not have a large impact on their results.

To evaluate this case, they have generated a dataset with the PhlatCam setup. We refer to the induced ML model as PhlatNet.

3.2.3 Model architecture

As the Monakhova et al. approach, the FlatNet approach also uses the U-Net after their inversion layer, but with a different depth and number of stacked convolutions. Multiple models were tested, namely U-Net$_{32}$, U-Net$_{64}$ and U-Net$_{128}$ where the index refers to the first number of filters of the first stage. More precisely we have the following number of filters for each model, from the smallest to the largest model:

1. U-Net$_{32}$: [32, 64, 128, 256]
2. U-Net$_{64}$: [64, 128, 256, 512]
3. U-Net$_{128}$: [128, 256, 512, 1024]

There are also only two stacked convolutions for both the encoder and the decoder. Another major difference is that no max-pooling layers are employed, but instead to reduce the dimension they have for the first convolution in each stage a stride of 2, cutting the height and width size by two as the max-pooling would do.

3.2.4 Losses and discriminator

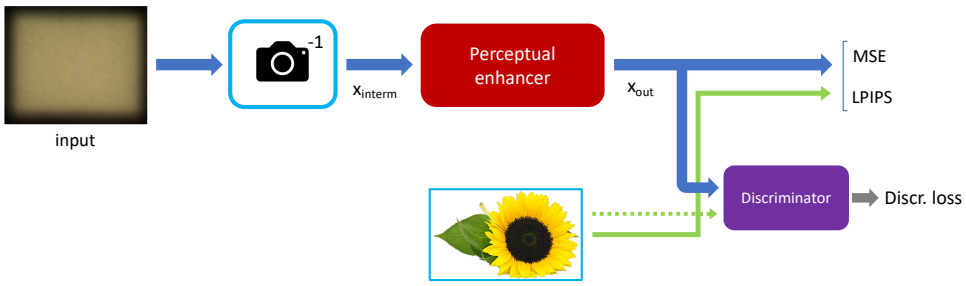


Figure 3.3: FlatNet architecture and process

.

The FlatNet approach also utilizes the MSE and the LPIPS loss, but in addition, it is constructed as a conditional generative adversarial (cGAN)[15]. To recap this technique, there are two ML models: on one side the generator, here our reconstruction model, and on the other side the discriminator. The goal of the discriminator is to differentiate between a real image (here the ground truth image), and the image generated by the generator, and the goal of the generator is to fool the discriminator. We train them together in a concurrent way. This framework is commonly used to generate realistic images from noise. Here, the "conditional" of cGAN means that the generator has more insights on what type of image it has to generate. In our case, the conditional part is simply the measurement sample, as showed in Fig. 3.3.

The discriminator model is a CNN with 4 blocks containing a convolution of a kernel size of 3×3 followed by a swish activation[17], defined as :

$$\text{swish}(x) = x\,\text{sigmoid}(\beta x) = \frac{x}{1+e^{-\beta x}},$$

then an average pooling, and finally a convolution with a kernel size of 1 to obtain at the end only one value, either a probability between 0 and 1 if we add a sigmoid function at the end, evaluating how likely is the input to be real, or the logit of this probability without the sigmoid. For training, the discriminator has its own loss :

$$\mathscr{L}_{\text{disc}} = -\log\left(D\left(Y_{\text{true}}\right)\right) - \log\left(1 - D\left(Y_{\text{gen}}\right)\right),$$

where D is the discriminator model and Y_{gen} the output of the generator. The generator needs also to take into account the discriminator output to be able to mislead the discriminator in the future. It is done with another term called the adversarial loss:

$$\mathscr{L}_{\text{adv}} = -\log(D(Y_{\text{gen}})).$$

As the LPIPS loss, this adversarial loss also enforces the model to generate more realistic results for human perception. In their proposed GAN implementation, they use label smoothing [21] to regularize the discriminator loss.

Chapter 4

Model optimization

When we want to deploy a ML model to an embedded device, one has to handle multiple challenges due to the hardware constraints of the targeted device, and also the given application. Firstly, for most edge devices, all machine learning frameworks such as TensorFlow or PyTorch are not supported as it. Indeed, most of such devices, such as microcontrollers, do not have a very developed operating system and a proper set of operation to handle such frameworks and support all operations that can be done on a CPU. The reason depends on for what purpose the device has been designed: low-memory footprint, low-cost, low energy consumption. All these constraints are not taken into account when we train a ML model on a server or on a computer. For example, in basic microcontrollers, the available RAM are only a few kilobytes, orders of magnitude smaller than a basic computer; the same for the flash memory, the computational power, etc... To be able to deploy any model to an edge device, we use the TensorFlow Lite library.

To reduce the model for memory efficiency, accelerate the inference time, we have explored various techniques.

4.1 Quantization

In our context, quantization implies converting a 32 bits floating point value stored into an 8 bits integer value. More specifically, we quantize the weights of our model, what we call "quantized weight". Furthermore, if needed, one can also do the operations in the integer form, what we call "Quantized inference". To explain a bit more the last point, in a "quantized weight", we compress the weights in an 8 bits representation and store them. When we have to compute the operation with the input, which is in a floating point representation, we decompress the weights and do the computation in the floating point representation. This method reduces the size of the model by a factor of (roughly) 4, and also reduces the bandwidth of moving data from flash to RAM. But the

RAM memory consumption during inference does not change since we still do the computation with floats. Furthermore, some microcontrollers do not support floating-point multiplication. For these reasons, the "quantized inference" makes a step further by not only quantizing the weights, but also performing the operations in the integer representation as follow :

$$output = quantize(input, step) * quantize(weight, step),$$

where

$$quantize(x, step) = round(x * step)/step$$

and with $step$ handling the number of decimals we want. So setting it to 1 returns the nearest integer. Naturally, all quantization techniques come with an accuracy cost, since conversion from a floating point value to an integer value is not for free. There is also the possibility to not quantize everything in the model, what we call "Hybrid quantization", in opposition of "Pure/Full integer quantization". For instance, we can keep only the biases and activations in their 32 bits floating point format, because practically these values are very sensitive to quantization, and also in terms of memory footprint their number is very low compared to the number of weights in general, so storing them with 32 bits does not dramatically increase the overall model size. For convolutional models, hybrid quantization can speed up the inference time from 10 to 50%, and pure integer quantization around 50%. One can quantize the model after having trained it, what we call post-training quantization, or consider it during training, what we call Quantization Aware Training (QAT).

4.1.1 Post-training quantization

In the quantized weight case, to compress the weight value, we need to scale the floating point value to the 8 bits integer, we therefore need to find the minimum and maximum floating value encountered as depicted in Fig. 4.1.

4.2 Pruning

The idea of pruning is to remove unnecessary weights, i.e. weights that do not have a big influence on the output. Practically, the weights in our case can be seen as a matrix, and pruning is to force some of these values to be zero. We get at the end a sparse matrix that can be efficiently stored with a reduction of the model. There exists multiple ways to do this pruning. The most common one is the **magnitude pruning**. It consists of setting to zero the smallest weights in the network. Formally, we have this formula :

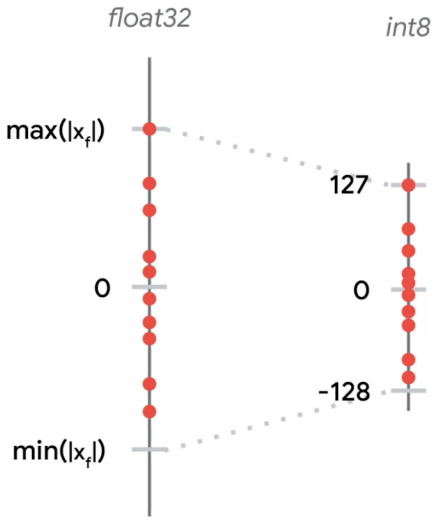


Figure 4.1: Scale float and integer representation

$$w_i = \begin{cases} w_i : \text{ if } |w_i| > \lambda \\ 0 : \text{ if } |w_i| \leq \lambda \end{cases}$$

Where λ is a threshold parameter that adjust how much we want to prune the model. A clever variant called **sensitivity pruning** is to set λ to $s * \sigma_l$ where σ_l is the standard deviation of the layer l measured in the dense model and s a tunable parameter. On top of that, instead of considering individual weights, one can consider groups of weights such as channels or filters. This is **structured pruning**. As an example, in some cases, magnitude pruning can reduce a model size up to 6 times with a negligible loss of accuracy. Pruning can be done after training, offline, but the most efficient way is to retrain a model and to consider pruning at this moment.

4.3 Clustering

Another way to compress a ML model is instead to limit the number of possible values. These values are the centroids of each possible clusters. In the weight matrix, we replace then each value with the index of the closest centroid. By doing so, we can also reduce the model size up to a factor 6 by setting the number of clusters to a small value. As pruning, it is better to retrain a pre-trained model and consider clustering within the training process.

4.4 Collaborative optimization

Collaborative optimization consists of combining the above methods one after the other to add all the benefits of each technique while still keeping a good overall model. The main collaborative methods are **PQAT**, **CQAT** and **PCQAT**, where **P** and **C** stand for pruned and clustered respectively. Their processes follow the same steps. For instance, PCQAT first takes the original trained model, retrains with pruning, then retrains the resulted model with clustering while maintaining what has done the pruning method, and finally retrains again the resulting model with the quantize aware training method while maintaining the work done by the two previous steps.

Chapter 5

Design and implementation

Practically, when evaluating the Phlatnet models, we faced GPU memory issues.

5.1 General considerations

The provided code is aimed to take the advantages of the GPU use. Each possible component of the ML model was made to be modular, that is, one can easily add, change or remove the discriminator, the camera inversion, the U-Net for a given dataset.

All the project was written in Tensorflow even though the original papers studied were implemented with PyTorch. This extra work needed was motivated by multiple reasons. There exists the PyTorch Mobile[1] to deploy models, but supports only IOS, Android and Linux operating systems, whereas Tensorflow Lite could be deployed into any microcontroller and is globally a more mature framework. It is possible to convert a PyTorch model to a TensorFlow graph by passing through the ONNX[2] format. But passing through this format could yield to another major problem : The set of possible operators in TensorFlow Lite is a subset of all the operators in Tensorflow, and TensorFlow Lite Micro (for microcontrollers) a very constrained subset of TensorFlow Lite. During the conversion, the current version of ONNX optimizes the given model operations used beneath, and can introduce some operators unknown to the TensorFlow Lite Micro set of operations.

Still special functions, such as the inversion of the Fourier transform, are not supported by TensorFlow Lite at this date, but could be available in the future, and already implementing the model in TensorFlow facilitates the integration and also keep the compliance in any future scenarios. Currently, it is possible to do any model optimization (QAT, PQAT,...) with the camera inversion layer, but for now, for inference, the camera inversion has to be handled differently, outside the Tflite library. Practically, for the moment, this is done by loading the weights into a Numpy matrices and

the layer computation performed in Numpy. The last reason to use TensorFlow is that it provides a large variety of possible model optimizations not available in PyTorch, such as clustering.

Due to the overall RAM memory consumption, one has to reduce drastically the batch size, leading to increasing the training time. To attempt to resolve this issue, one has tried to rewrite the code so that it is able to training with a multi-GPU setup. Unfortunately, even though we could observe a significant speed-up, the validation loss we finally obtained tended to diverge, in contrary to the training loss. We have therefore decided to train all our models with only a single GPU.

Both papers have utilized a learning rate scheduler to reduce the learning rate of the optimizer, that is, after a fixed number of epochs or batch step, we multiply the learning rate by a factor below 1. In our configuration, we have preferred to employ another mechanism reducing the learning rate only when the loss does not improve after a certain amount of epochs, because it generates better loss values.

The weights of theLPIPS model were trained in PyTorch, therefore to be exploitable in a TensorFlow project, we have converted it via ONNX to get a TensorFlow executable graph. One has also to be aware that the images feeding into it range lies between -1 and 1 instead of 0 and 1, otherwise the value outputted is lowered with respect to the real value.

5.2 Monakhova et al. approach

In their paper, it is said that they modify the factors of the MSE and LPIPS in the loss, reducing the MSE factor and augmenting the LPIPS factor as the epoch pass. But it is not explained how, if this was done by adding or multiplying the factor by a certain value. We have decided to include the possibility to add and subtract with the same value the corresponding loss factor. Nevertheless, in our experiments we have kept these values fixed as in FlatNet[11]. Our motivation is when the MSE validation reaches a very low, the ML model training loss is naturally driven by the other term, the LPIPS loss. Also, in their evaluation, the ground truth samples are not cropped to have only the picture of the screen, and thus having a large proportion of the image being only dark. Therefore, the evaluation metrics will give better results, since it will be straightforward for the ML model to localize the dark area and always predict an intensity of zero for all this area.

5.3 FlatNet

In their paper, they do not use the official version of the LPIPS metric, but rather a simplification of it: they only consider the output of two layers rather than five as the original, and simply sum their l_2 distance without reweighing the two terms. We have decided to only consider the official version in our implementation, so that it becomes more meaningful and easier to compare across

completely different lensless camera setups with the same standard.

When training with the corresponding datasets, more particularly with the PhlatCam one, we faced very frequently RAM consumption issues. Both datasets (FlatCam and PhlatCam) raw measurements were stored in their Bayer representation, with a dimension of respectively $512 \times 640 \times 4$ and $1518 \times 2012 \times 4$. In a setup where we have to store in RAM the computation of three models, of the reconstruction model, the LPIPS model and of the discriminator, it was already barely feasible with the FlatCam with the U-Net$_{64}$ (so not even the largest model) a batch-size of only 4 samples, and with 12 GB of GPU memory. Therefore, in our experiments, we had to reduce drastically the measurements dimensions. For the FlatCam dataset, we have cropped the dataset in the same way the code they provide to get a dimension of $500 \times 620 \times 4$. For the PhlatNet dataset, we converted the measurements to RGB data, and downsampled by 4, leading to a shape of $759 \times 1006 \times 3$. Having RGB data instead of Bayer allow us to straightforwardly resize our images to shrink the shape even more.
In their implementation, for PhlatNet, they crop their dataset to shape and then pad the samples in the inversion layer. Doing a zero-padding so that the convolution in the Fourier domain remains valid in the borders of the image, that is augmenting the size of the image by 3 was not conceivable. So we kept the data uncropped, but we don't pad the data in the inversion layer.
For the non-separable inversion layer in 3.2.2, we have explored ways not to learn W in the spatial domain, but rather in the frequency domain, that is $W_{\mathscr{F}} = \mathscr{F}(W)$. The motivation behind this was to decrease the training time by not having to compute the Fourier transform. The first issue is that we have to store the real and imaginary part of each variable, increasing the memory by a factor 2. The second one was that, if we decide to pad the data to get a valid convolution in the Fourier domain, we increase the shape by 3, and thus we would increase the total number of variables by a factor 27 for RGB images (since we have three channels, and each channel having a padded image of 3×3 the original image), so unscalable for training. But this idea could be implemented for inference, that is, after having trained W, store directly $\mathscr{F}(W)$ to not have to recompute at each reconstruction this Fourier transform.

We have faced multiple training issues while training with GANs, having the discriminator and adversarial losses remaining quickly constant over the epochs. We have first tested with two fully independent optimizers for the generator and the discriminator. After a few tests, we found that it was better that the learning rate of both optimizer remain equals.

Chapter 6

Implementation and Experiments

Here we present the results we have generated. We first describe the setup utilized, then analyze the results compared with the paper's work we have implemented, and finally show an analysis of model optimization for deployment.

6.1 Setup

During all the evaluation, we have used the following setup. The Adam optimizer was used both for the reconstruction model and the discriminator model with a learning rate of 10^{-3} (each model has its own independent optimizer). On top of that, we reduce the learning rate by half when the corresponding loss stagnates with a minimal learning rate of $6 * 10^{-8}$. By "stagnates", we mean that in a window of 4 epochs, if the loss function has not reduced by at least 10^{-4}, then we reduce the learning rate by half. We have trained all the models for 75 epochs. We split the dataset into two sets, 80% for the training, and 20% for testing. We do not split further the testing set into two sets, for validation and test, since we do not use the test set for fine-tuning our model, as we would normally do in a standard ML task, since the goal is only to make the two paper results reproducible.

For the LPIPS loss, we have only considered the VGG based version, as performs better than the AlexNet one. When considering the weights of the losses, we have kept the same weights as the PhlatNet paper : $\lambda_{mse} = 1$, $\lambda_{lpips} = 1.2$ and $\lambda_{discr} = 0.6$, in the case where we do not use the discriminator during training, we set the λ_{lpips} to $1.2 + 0.6 = 1.8$ to enforce the same way the loss for the perception. Concerning the dataset processing, for the phlatnet dataset, we have chosen to center-crop each image like stated in their paper. We do the same for the PSF since our reconstruction model expects that the raw measurements has the same shape as the PSF, which should be the case.

In the PhlatNet paper for the non-separable case, the intermediate result given by the camera inversion is given by:

$$X_{\text{interm}} = \mathscr{F}^{-1}(\mathscr{F}(W) \odot \mathscr{F}(Y))$$

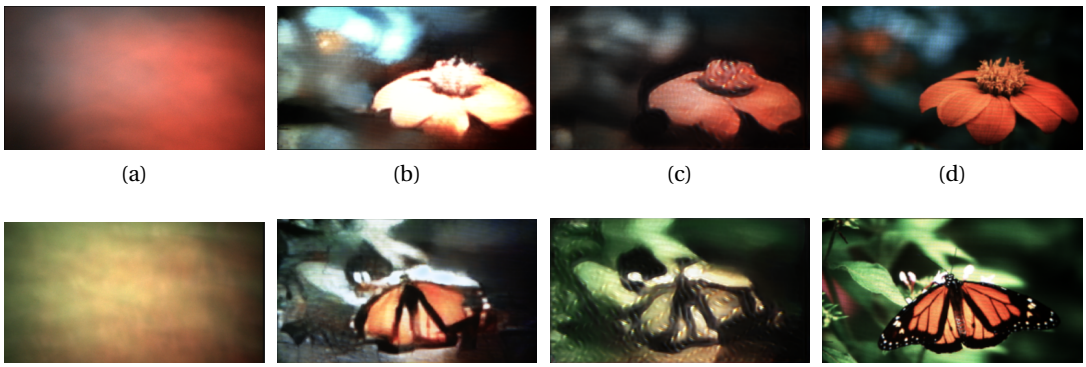
Where W is a trainable matrix. But practically, computing the Fourier transform is expensive during training. So instead, we study if learning directly the Fourier transform of W within the model, i.e. learning the matrix $W_f = \mathscr{F}(W)$. We expect to get a runtime improvement, in the training and in the inference time. We also evaluate if this approach reduces the accuracy of the model. Furthermore, if we choose to pad the matrix, then the resulting matrix will have more parameters.

6.2 Reproduction results

6.2.1 Monakhova et al. comparison results

we first compare the results we obtained compared to the results in the paper using the AlexNet LPIPS as their paper. Also, just for this case, we do not crop the ground truth samples to get only the display screen to compare faithfully with the paper results. As observed in Fig. 6.1, the MSE value we have obtained is greater than the one of the paper, in contrary to the LPIPS value which is quite lower. LPIPS ranges over 0 and 1, 0 meaning that the two images are very similar. This can be explained by the weights of the loss terms. Also, it is important to note that for our evaluation, as said previously, we use 20 percent of the dataset for testing, in contrary to the paper taking only 4 percent from the total dataset, a clearly weaker statistical value and not representing robustly the true performance of the model. We also show the results where we crop the ground truth images to get only the display screen. As a matter of course, both errors increase, since the cropping area, containing only zeros and thus easily identifiable for any ML model, corresponds 38.4% of the original image. From now on, we only employ cropped data. The reconstruction images are shown in 6.1 together with the reconstruction showed in the original paper. We can see that our implementation can reconstruct faithfully some images, but as we look more closely, some artefacts appear, especially for the butterfly image. This could be due to the fact that in training we may be weighting the MSE and LPIPS losses differently than in the original paper (they do not publish how they do it).

As showed in Fig. 6.2, the validation accuracy tends to be unstable in the first epochs, due to the larger learning rate we employ at the beginning to converge faster. The proposed model tends to create a significant difference between the training LPIPS loss and the validation one, with a minimal training loss value of 0.2551 compared to 0.3143. This could mean that there exists a gap of improvement, either by changing the model architecture, or by adding weight regularization, such as the L_1 norm.



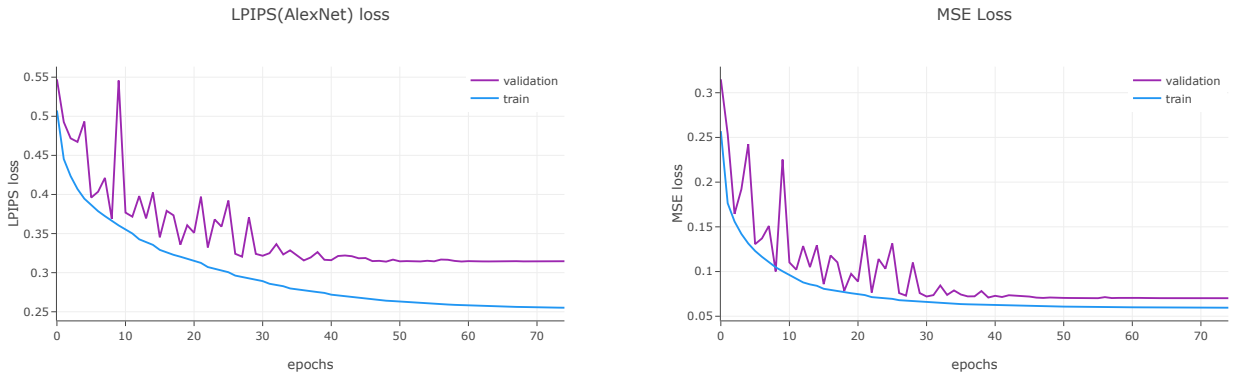
(a) (b) (c) (d)

(e) Raw measurements (f) Expected reconstruction (g) Current reconstruction (h) Ground truth

Figure 6.1: Results of Monakhova et al. U-net over DiffuserCam.

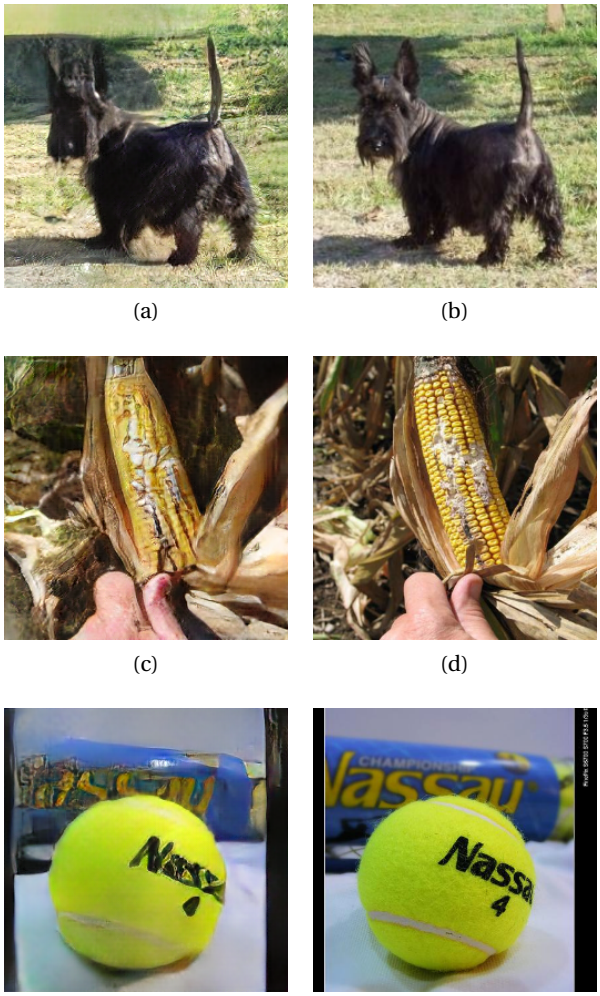
	Original	Ours (not cropped)	Ours (cropped)
Mean-squared error	**0.0154**	0.0379	0.0702
LPIPS (Alex)	0.2461	**0.2314**	0.3143

Table 6.1: Comparison of results of U-Net$_{\text{waller}}$ over DiffuserCam



(a) MSE loss (b) LPIPS(AlexNet)

Figure 6.2: Results of Wallerlab with cropping



(a) (b)

(c) (d)

(e) Proposed reconstruction (f) Ground truth

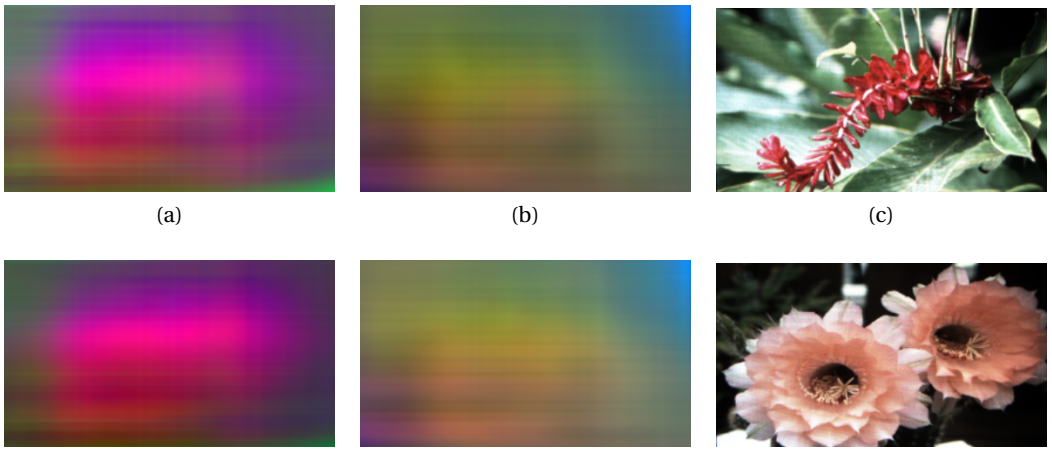
Figure 6.3: Reconstruction for FlatNet.

6.2.2 FlatNet comparison results

As said previously, we have preferred to employ the standard VGG LPIPS loss rather than their proposed (unconventional) loss, therefore we do not compare the LPIPS values we obtained with theirs. For the separable case, the results are showed here in Table 6.2. For both SSIM and PSNR, an higher value demonstrates better results. The results are similar, even getting a better PSNR for our model. The difference of SSIM can be explained by the change of LPIPS function. We show here Fig.6.3 some reconstructions we have obtained. They look surprisingly well compared to the Monakhova et al. approach. But one needs to remember that the ground truth images in this dataset were not captured by a lens camera, but are the original image files.

	Original	Our implementation
PSNR (in dB)	19.62	**19.80**
SSIM	**0.64**	0.53

Table 6.2: Comparison of results for FlatNet dataset



(a) (b) (c)

(d) Simple model (e) Pruned model (f) Ground truth

Figure 6.4: Camera inversion intermediate output over DiffuserCam

But for the non-separable with the PhlatCam dataset, we were unable to reproduce values even approaching theirs. Multiple reasons can be presented, such as how we preprocess the datasets for memory allocation purposes (downsampling, transforming to RGB, resizing), or the differences or implementation, like the LPIPS loss function. Nevertheless, we have applied the ideas of their proposed inversion layer in the following section, and show its benefits.

6.3 Comparison of different approaches

For this section, we evaluate different ML models/frameworks only with the DiffuserCam dataset. We do not use label smoothing for the discriminator. As the table Fig. 6.3, it is clear that the inversion layer3.2.2 helps greatly to improve the output (the inversion layer used was the one for the non-separable case). But the assumption that the inversion layer is trying to invert the forward model explicitly does not hold, as shown in Fig. 6.4: we expect to see an intermediate image resembling the ground truth, which is clearly not the case. We evaluate here two different models based on U-Net$_{\text{wal-64}}$ that is introduced later (the simple model, and the pruned), only to show that the intermediate result also greatly change across different models.So this layer mainly helps the model by augmenting its representation power. Indeed, practically, the number of weights within the inversion layer represents almost 40% of the total. The results using the discriminator do not show

U-Net	Simple	Only discr.	Only inversion	Discr. + inversion
Parameters	11,784,211	11,784,211	18,005,012	18,005,012
PSNR (in dB)	18.5271	16.8318	**19.6237**	16.4251
SSIM	0.5128	0.4876	**0.5597**	0.4822
LPIPS (VGG)	0.4606	0.4821	**0.4322**	0.4900
MSE	0.0676	0.1024	**0.0528**	0.1056

Table 6.3: DiffuserCam results with U-Net (Monakhova et al.)

distinctly real benefits. It is relevant to say that in general, training GANs can be extremely unstable, depending on the discriminator, the generator architecture and other hyperparameters. Here we have employed the same discriminator architecture as the FlatNet paper, and thus could not be perfectly suited for the DiffuserCam dataset.

We also compare the results with another model that has the same architecture as the U-Net$_{\text{waller}}$, with the difference that the list of filters is the same as U-Net$_{64}$, with the idea to see if changing slightly the model to look like the U-Nets of the FlatNet. We also add to it the non-separable inversion layer. We refer to this model as U-Net$_{\text{wal-64}}$.It gives us the best results so far, obtaining correspondingly the following values:

- PSNR: 21.0883
- SSIM: 0.6639
- LPIPS: 0.3779
- MSE 0.0386.

Some graphical results are showed in the following section, where we compare the output with other models.

6.4 Model optimization results

During this section, we have decided to use the U-Net$_{\text{wal-64}}$, as it gave best results. One of the reasons of not using directly to U-Net$_{64}$ was to avoid the PixelShuffle part, as it could be not compliant with the Tensorflow Lite Micro library available operators. We then evaluate over this model how model optimization techniques could change the model performance, and what are the gains in terms of runtime and memory.

U-Net	Normal	Clustered 128 centroids	Pruned	QAT
LPIPS (VGG)	0.3779	0.7478	0.4047	0.4642
PSNR	21.0883	9.332	20.23	17.08
SSIM	0.6639	0.2978	0.624	0.5068
MSE	0.0386	0.5008	0.04495	0.0953
Disk storage (MB)	62.682	16.741	32.171	38.72

Table 6.4: DiffuserCam and model optimization results

6.4.1 Accuracy

We first evaluate how the given model performs in terms of accuracy/loss. As showed in Fig. 6.4, we compare the results where the first model (Normal) is the model without any modification and memory constraints, then a model trained with clustering, a model trained with pruning, and a last model trained with QAT. We also show the FLASH memory they use in TensorFlow. We have decided to not evaluate any collaborative optimizations methods, as it would even more deteriorate the results. As the results show, we suffer from a loss for each metric and each model. For the clustered model, the results are not even usable. As told previously, 128 centroids mean 128 different values for each layer, which is in this case not sufficient. We were unable to increase this amount due to RAM consumption and how clustering training works. The pruned case was the one having the best values among the optimized models. Looking at the disk storage, we can observe a decrease of memory by a factor 2: here the model was trained to have at the end 50% of sparsity, so half of the weights being zero. The QAT model does not perform very well, but still decently regarding how is built, as we will show later. Its disk storage in TensorFlow is an upper bound on the real value, since the TensorFlow models contains also all layer wrappers to be able to train the model with QAT.

Now, for the illustrations showed in Fig. 6.5, we compare various cases: The normal case, where the weights are not quantized, the second case where the weights are post-quantized, but the operations are kept in floating-point, and the last case where the weights are post-quantized and operations are made in 8 bits. As a reminder, QAT trains a model so that it can the weights can be quantized, and the operations in 8 bits without deteriorating too much the results. As a remark, we do not quantize the inversion layer, as it is outside the Tflite model (since some operators are not supported, as told previously). As the picture shows, quantizing the weights degrades slightly the image, but the results are still good overall. But the problem is when we want to do also the operations in the integer domain: both the pruned and the normal model can simply not reconstruct the images at all, in contrary to the QAT model that was train particularly for this case, and thus giving usable results.

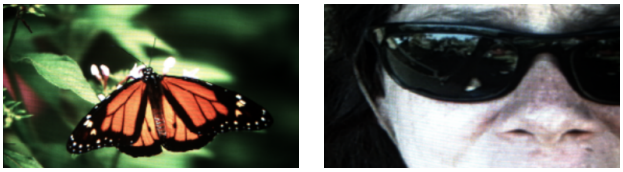
	File size (MB)	RAM (MB)	Init. time[us]	Inference time[us]
Normal	44.15	316.65	112600	558700
Normal + quant	11.11	771.75	200900	521900
normal + quant + op.	11.11	134.0	63800	699300
Camera inversion	24.9	285.0	70000	3110000

Table 6.5: Memory and running time of perceptual model through optimization

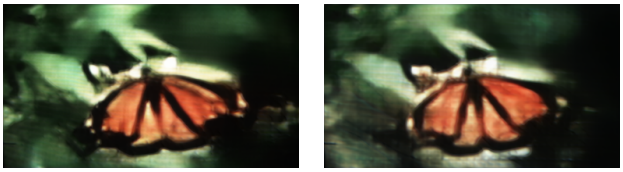
6.5 Comparison of different approaches

6.5.1 Memory and running time

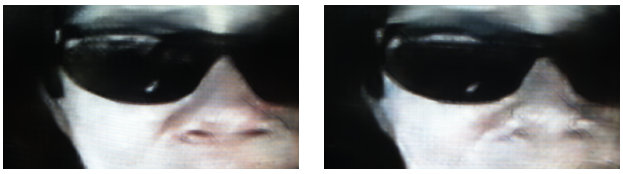
The results for Tensorflow Lite were generated using the benchmark provided directly by Tensorflow. We show here multiple inference times with 8 threads, as it corresponds to the number of cores that have most of the mobile phones. The table shows only the perceptual part, the U-Net, as only this part can be quantized in Tflite. We evaluate the camera inversion layer separately: it is not part of the quantized Tensorflow Lite model. We only evaluate it by converting to another Tflite model separately that can support some Tensorflow operations for completeness (but that cannot be deployed to most of the devices, and cannot be quantized, just for analysis purposes). The results are showed in Fig.6.5. We only evaluate the "normal", i.e. the unmodified U-Net$_{\text{wal-64}}$. The As expected, quantizing the weights reduces by a factor 4 the file size of the Tflite model. But surprisingly, the model performing in floating-point value, but with quantized weights has a larger RAM memory consumption than the unquantized one. This is probably do to the fact that is has to allocate memory to convert back these weights from integer to floats. Therefore, the initialization time is larger, as it has to allocate more memory. The model with quantization and computing with integer operations use therefore the less RAM. It is also relevant to look at the inference time of the camera inversion layer: it is almost 5 times larger compared to the others, due to the Fourier transforms it has to perform.



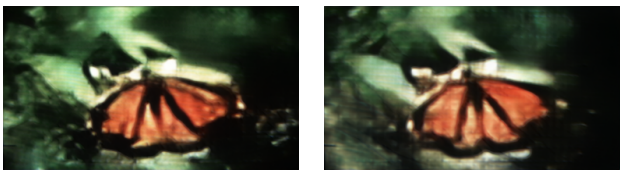
(a) Ground truth (b) Ground truth



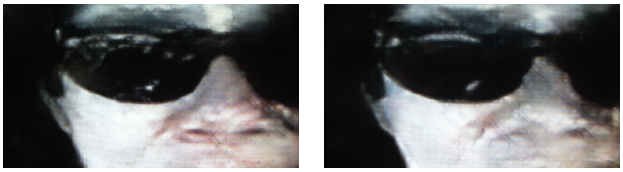
(c) (d)



(e) Normal (f) Pruned



(g) (h)



(i) Normal + Quant. (j) Pruned + Quant.

(k) (l) (m)

(n) Normal + Quant. + Op-quant. (o) Pruned + Quant + Op-quant. (p) QAT

Figure 6.5: Results of model trained with optimization

Chapter 7

Going further

We have explored various ways to reduce the model size with techniques such as pruning, quantizing or clustering, but there exists another approach that could be relevant to study, radically different, called Knowledge Distillation. The idea is to begin with a trained model that has a good accuracy that we call the "teacher network", and to build another smaller model called the "student network" that is trained to mimic the teacher network output.

Other more advanced U-Nets models were incorporated into the project and tested properly thanks to the work provided in [19], but due to lack of time, they were not extensively evaluated. Evaluating them could improve the performance of these models, as it is not clear at this date which architecture is the most suited for lensless image reconstruction.

To be even more memory efficient in the PhlatNet dataset, one could consider converting the Bayer measurements to RGB before training, in other words processing 3-channels data instead of 4-channels data, reducing the model size by about 25%. It is unclear for now how much the model accuracy decreases when doing so.

Currently, it is clear that none of the models are neither sufficiently small or efficient to be deployed into microcontrollers. Due to the task itself, the model has to produce a large and detailed output. This constraint limits the model size reduction, as well as the performance. But it does not hold for all ML tasks: one could think of applications like image classification, where we could port more robust Tflite models into smaller embedded systems.

As showed in the results, the camera inversion layer proposed by FlatNet does not really provide an intermediate result that looks similar to the ground truth in any ways when applied to the DiffCam dataset. To really enforce this property to hold, and also improve the results based on the theory behind this idea, one can consider computing also the loss over this intermediate representation and do deep supervision. This idea can also be applied for the Monakhova et al. paper where they

use ADMM-based models with a U-Net.

Chapter 8

Conclusion

In this thesis, we have implemented the Monakhova et al. approach[16], as well as the FlatNet paper[11]. We were able to reproduce the results of the first paper, and even improving them by using a slightly different U-Net. We were also able to generate convincing results for the FlatCam dataset used for the separable case presented in the FlatNet paper, but not for PhlatCam, the non-separable case. We have showed that the inversion layer applied in the dataset used by Monakhova et al. was beneficial and improves consequently the model. Nevertheless, when studying the output at this stage, i.e. before the U-Net, it does not show clear characteristics of the ground truth image, and its representation varies across different architectures.

For the model optimization part, we have analysed various model optimization techniques, so that the model can be efficiently deployed. QAT and weight pruning can be used for reconstruction, which is not the case of the clustering technique. The resulted memory consumption could be handled by mobile phones, but hardly for most microcontrollers. The disk storage can be lowered by a factor 4 with an acceptable reduction of accuracy when performing the operation in floating-point arithmetic. Having a model performing integer arithmetic to reduce the RAM consumption involves a large cost in terms of model accuracy, and can only be done with a model trained with QAT.

Bibliography

[1] URL: https://pytorch.org/mobile/home/.

[2] URL: https://onnx.ai/.

[3] M. Salman Asif, Ali Ayremlou, Aswin Sankaranarayanan, Ashok Veeraraghavan, and Richard Baraniuk. *FlatCam: Thin, Bare-Sensor Cameras using Coded Aperture and Computation*. 2016. arXiv: 1509.00116 [cs.CV].

[4] Eric Bezzam, Martin Vetterli, and Matthieu Simeoni. *Privacy-Enhancing Optical Embeddings for Lensless Classification*. 2022. arXiv: 2211.12864 [cs.CV].

[5] Vivek Boominathan, Jesse K. Adams, Jacob T. Robinson, and Ashok Veeraraghavan. "PhlatCam: Designed Phase-Mask Based Thin Lensless Camera". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.7 (2020), pp. 1618–1629. DOI: 10.1109/TPAMI.2020.2987489.

[6] Vivek Boominathan, Jacob T. Robinson, Laura Waller, and Ashok Veeraraghavan. "Recent advances in lensless imaging". In: *Optica* 9.1 (Jan. 2022), pp. 1–16. DOI: 10.1364/OPTICA.431361. URL: https://opg.optica.org/optica/abstract.cfm?URI=optica-9-1-1.

[7] Jieneng Chen, Yongyi Lu, Qihang Yu, Xiangde Luo, Ehsan Adeli, Yan Wang, Le Lu, Alan L. Yuille, and Yuyin Zhou. *TransUNet: Transformers Make Strong Encoders for Medical Image Segmentation*. 2021. arXiv: 2102.04306 [cs.CV].

[8] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. 2017. arXiv: 1606.00915 [cs.CV].

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[10] Michael J. DeWeert and Brian P. Farm. "Lensless coded-aperture imaging with separable Doubly-Toeplitz masks". In: *Optical Engineering* 54.2 (2015), p. 023102. DOI: 10.1117/1.OE.54.2.023102. URL: https://doi.org/10.1117/1.OE.54.2.023102.

[11] Salman Siddique Khan, Varun Sundar, Vivek Boominathan, Ashok Veeraraghavan, and Kaushik Mitra. "FlatNet: Towards Photorealistic Scene Reconstruction from Lensless Measurements". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020), pp. 1–1. DOI: 10.1109/tpami.2020.3033882. URL: https://arxiv.org/pdf/2010.15440.pdf.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[13] Grace Kuo, Nick Antipa, Ren Ng, and Laura Waller. "DiffuserCam: Diffuser-Based Lensless Cameras". In: *Imaging and Applied Optics 2017 (3D, AIO, COSI, IS, MATH, pcAOP)*. Optica Publishing Group, 2017, CTu3B.2. DOI: 10.1364/COSI.2017.CTu3B.2. URL: https://opg.optica.org/abstract.cfm?URI=COSI-2017-CTu3B.2.

[14] Grace Kuo, Fanglin Linda Liu, Irene Grossrubatscher, Ren Ng, and Laura Waller. "On-chip fluorescence microscopy with a random microlens diffuser". In: *Opt. Express* 28.6 (Mar. 2020), pp. 8384–8399. DOI: 10.1364/OE.382055. URL: https://opg.optica.org/oe/abstract.cfm?URI=oe-28-6-8384.

[15] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: 1411.1784 [cs.LG].

[16] Kristina Monakhova, Joshua Yurtsever, Grace Kuo, Nick Antipa, Kyrollos Yanny, and Laura Waller. "Learned reconstructions for practical mask-based lensless imaging". In: *Optics Express* 27.20 (Sept. 2019), p. 28075. DOI: 10.1364/oe.27.028075. URL: https://arxiv.org/pdf/1908.11502.pdf.

[17] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions*. 2017. arXiv: 1710.05941 [cs.NE].

[18] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[19] Yingkai Sha. *Keras-unet-collection*. https://github.com/yingkaisha/keras-unet-collection. 2021. DOI: 10.5281/zenodo.5449801.

[20] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[21] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[22] Xintao Wang, Liangbin Xie, Chao Dong, and Ying Shan. *Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data*. 2021. arXiv: 2107.10833 [eess.IV].

[23] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. arXiv: 1311.2901 [cs.CV].

[24] Kai Zhang, Yawei Li, Wangmeng Zuo, Lei Zhang, Luc Van Gool, and Radu Timofte. "Plug-and-Play Image Restoration with Deep Denoiser Prior". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.10 (2021), pp. 6360–6376.

[25] Kai Zhang, Jingyun Liang, Luc Van Gool, and Radu Timofte. "Designing a Practical Degradation Model for Deep Blind Image Super-Resolution". In: *IEEE International Conference on Computer Vision*. 2021, pp. 4791–4800.

[26] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric*. 2018. arXiv: 1801.03924 [cs.CV].

[27] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. *UNet++: A Nested U-Net Architecture for Medical Image Segmentation*. 2018. arXiv: 1807.10165 [cs.CV].

[28] Wangmeng Zuo, Kai Zhang, and Lei Zhang. "Convolutional Neural Networks for Image Denoising and Restoration". In: *Denoising of Photographic Images and Video: Fundamentals, Open Challenges and New Trends*. Ed. by Marcelo Bertalmío. Cham: Springer International Publishing, 2018, pp. 93–123. ISBN: 978-3-319-96029-6. DOI: 10.1007/978-3-319-96029-6_4. URL: https://doi.org/10.1007/978-3-319-96029-6_4.