Tampere University

Vladimir Vashchenko

ANALYSIS OF SPARSITY- AND NONLOCALITY-REINFORCED CONVOLUTIONAL NEURAL NETWORKS FOR IMAGE DENOISING

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
March 2022

ABSTRACT

Vladimir Vashchenko: Analysis of Sparsity- and Nonlocality-Reinforced convolutional Neural Networks for Image Denoising
Master of Science Thesis
Tampere University
Major: Data Engineering and Machine Learning
March 2022

Different neural networks are built using the same elements, but they vary a lot by architecture and show different denoising quality results. The master thesis aims to analyze how different network hyperparameters, input transformations (sparsity), and nonlocal filters (nonlocality) impact the performance in image denoising tasks. The thesis work provides rich experimentally based research and analysis in the image denoising field. A number of different denoising methods have been considered, and state-of-the-art denoising algorithms were examined. As a result of the parameter analysis, the vast majority of the learning-based algorithms and deep networks were improved in terms of denoising capability.

Keywords: image denoising, image processing, image transforms, convolutional neural network, machine learning, deep learning

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

AWGN	Additive White Gaussian Noise
BM3D	Block-matching and 3D Filtering
BMCNN	Block-Matching Convolutional Neural Network
BN	Batch Normalization
CNN	Convolutional Neural Network
CNNF	CNN-based filter
CPU	central processing unit
DCT	Discrete Cosine Transform
DnCNN	denoising convolutional neural network
DWT	Discrete Wavelet Transform
FCN	Fully Convolutional Network
FFDNet	fast and flexible denoising CNN
FLCNN	FlashLight CNN
GPU	graphics processing unit
IDCT	Inverse Discrete Cosine Transform
IDWT	Inverse Discrete Wavelet Transform
JPEG	Joint Photographic Experts Group
KLT	Karhunen-Loeve Transform
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MWCNN	Multi-level Wavelet-CNN
NLF	Non-local Filter
NLM	Non-local Means
NLS	Non-local Self Similarity
NN3D	neural networks and 3D collaborative filtering
PSNR	Peak Signal-to-noise Ratio
ReLU	Rectified Linear Unit
RGB	Red Green Blue color model
SSIM	Structural Similarity

STFT	Short Time Fourier Transform
SVD	Singular Value Decomposition
SWT	Stationary Wavelet Transform
WCNN	Wavelet CNN
WNNM	Weighted Nuclear Norm Minimization

1 INTRODUCTION

With the wide adoption of digital cameras, the request for high-quality images for further applications is becoming more significant than ever. Digital pictures obtained on camera sensors are processed using algorithms such as demosaicing and denoising before their utilization. Compared to professional photography cameras, most modern devices including cellphones have smaller cameras with more inferior optics and digital sensors, which makes the quality of obtained images worse, especially in the challenging environment like at night time. However, when the proper pipeline of digital processing is applied to the data, the resulting images can be significantly enhanced, despite the property of the sensor. With more comprehensive capabilities and performance enclosed in modern smartphones and other devices, it becomes possible to run a variety of efficient algorithms right on a device. The whole image capturing pipeline more and more relies not just on lenses and sensors quality, but more on the software that performs a lot of sophisticated analysis and enhancements which make this called computational photography. Algorithms for filtering images from noisy measurements are of great interest. The fundamental problem known as image denoising will be addressed in this thesis. In this work by denoising, we understand the suppression of Additive White Gaussian Noise (AWGN) with zero mean from noisy images. The AWGN model is used as it behaves similarly to some real noises and therefore gained widespread adoption among the denoising algorithms.

There are different image denoising algorithms been developed and broadly utilized in the past years. Some of the non-learning signal processing approaches for image denoising are taken as benchmarks in overwhelming works in comparative analysis, such as K-SVD [1], WNNM [15], or BM3D [11]. The learning-based algorithms for image denoising started evolving rapidly due to the recent growth of computational power and widespread use of parallel computing with graphic cards and similar chips and neural engines for machine learning algorithms embedded even in small portable devices including modern smartphones. Recently learning-based models reached similar performance to the not learning-based algorithms. Learning-based methods serve as a black box. Providing examples of noisy input images and clean ground truth output images as the desired result, a network trains to remove the added noise. We will examine the most promising networks, such as DnCNN [32], FFDNet [33], MWCNN [21], and others.

Different neural networks are built using the same elements, but they vary a lot by architecture and show different denoising quality results. The number of parameters is significant in learning-based methods. That is why it is still not evident what affects the performance and how it can be further improved. In this work, we will analyze hyperparameters of the different networks to understand what specifically facilitates the denoising quality. We will also study how the input image itself can alter the denoising quality if provided in different dimensions, for example, if some sparsification transform is applied to the input. In addition, a combination of not learning-based methods and CNN filtering will be considered. Evaluation of the models is done quantitatively (Peak Signal-to-noise Ratio (PSNR) values are compared) and qualitatively.

The thesis is organized as follows. First, we study the fundamental concepts and basics of denoising algorithms, including current non-learning and learning-based techniques. Then we examine the hyperparameters for some of the learning-based methods to see what drives the performance. Then we will study how the sparsification of an input image changes the performances of the networks. Lastly, we will discuss and examine the effect of combining the non-local methods along with CNN filtering. The last chapter enclosed the conclusion of the work and further possible research.

2 IMAGE DENOISING METHODS OVERVIEW

This chapter addresses the development of image denoising methods to provide a comprehensive overview. Here we describe some proven and time-tested classical image filtering algorithms as well as state-of-the-art denoising techniques. In this chapter, we define the algorithms as they appear in the corresponding scientific articles, while a thorough analysis of the selected algorithms will be present in the following chapters.

First, we will explain the purpose of the denoising procedure in general. Then we proceed over some specific methods, following classification from a recently published denoising algorithms review [14], complementing it with some advanced deep learning-based techniques.

2.1 Image denoising problem statement

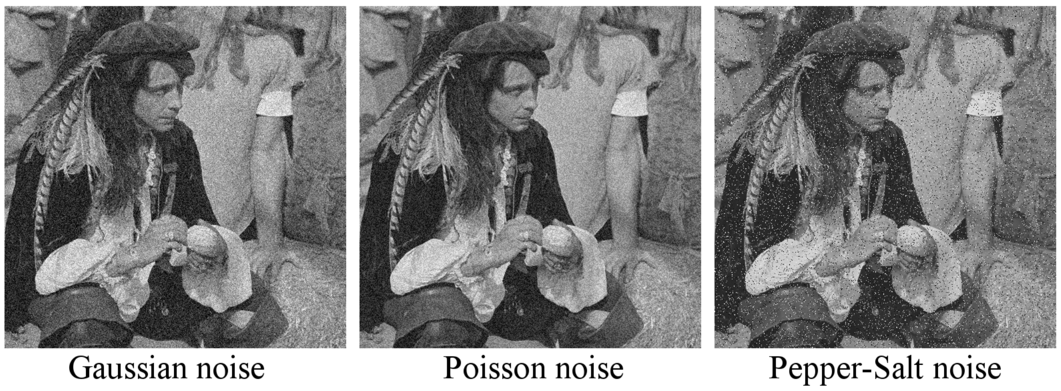

Figure 2.1. *Standard pirate test image corrupted by different types of noise: Gaussian, Poisson, and pepper-and-salt. [14]*

The goal of any image denoising algorithm regardless of its implementation is to reconstruct a high-quality image from its noisy observation. This is a mandatory step when dealing with images obtained on the camera sensors which makes this problem ubiquitous.

Image degradation model can be simply described with the following formula:

$$z(x) = y(x) + n(x),$$

where y yields to a clean image, z refers to a noisy image, n represents a noise model, and x is a 2D coordinate. In real life, a clean image from this formula does not exist alone and the image obtained on the camera sensor is already degraded by real noise. However, for computer modeling purposes instead of getting a degraded image from a sensor, we corrupt a known image that we consider a clean one with some noise model. Most of the research in the field is carried out with the Additive White Gaussian Noise model, considering noise to be signal independent and identically distributed with zero-mean and standard variance value σ^2: $n(x) \sim \mathcal{N}(0, \sigma^2)$. For this reason, Poisson noise and Salt-Pepper noise models are not considered due to the difficulties in comparison of the denoising results. The effect of using different noise models to contaminate an image is illustrated in Figure 2.1.

The difficulty of the image denoising task is that after the noise added to an image, the original pixel values are lost completely, which makes it a challenging inverse problem.

Peak signal-to-noise ratio

In order to evaluate how the image denoising problem is solved, or, in other words, how the estimated image differs from the original clean image that we have in the experiment setup, a quantitative characteristic needs to be introduced. When the estimate of a corrupted image is obtained, we calculate how close (similar) it is to the corresponding clean image using Peak Signal-to-noise Ratio (PSNR) characteristic, that is measured in dB and calculated using the following expression:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_Y^2}{MSE}\right), where$$
$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[Y(i,j) - \hat{Y}(i,j)\right]^2,$$

where Y refers to a clean image, $\hat{Y}$ to its noisy approximation, and MSE is the mean squared error between these two images. As we can see from the formulas, if the estimate is identical to the clean image, then the PSNR value tends to infinity, so the bigger the value the stronger is the similarity between the original clean image and its estimate. However, worth to notice that it is also possible to artificially make up such an image that is very, or even completely different from the original clean one, while the PSNR value can still be high. Nevertheless, PSNR represents well the quality of the algorithms, being the main widely-used characteristic used in literature.

As we now agreed to use PSNR as a quantitative characteristic of an algorithm's quality, we can calculate it for any image that has been recovered from noise comparing to the one we consider clean. However, for different images PSNR value will vary as its formula uses the maximum pixel value of a clean image. Moreover, the PSNR value for the same image may change, as the MSE value can fluctuate from one denoising model to another. Some algorithms are better in denoising specific image content, for example, repetitive patterns, structures, and textures. Other algorithms can perform better in denoising images with rather smooth content. To neglect this variation all of the algorithms, as a rule, are usually tested not on a single image, but on a dataset. The most commonly used dataset Set12 depicted in Figure 2.2 contains twelve black and white images both with rather smooth content (for example, Cameraman, Peppers, and Airplane) and with repetitive structured content (repetitive patterns of the brick wall in House, repeated round patterns in Starfish and Monarch, patterns on the tablecloth, headscarf and pants in Barbara). When testing on the dataset, the mean PSNR value of all twelve is obtained as a PSNR value of the algorithm. This mean PSNR usually compared to the results of the other algorithms performed on the same dataset.

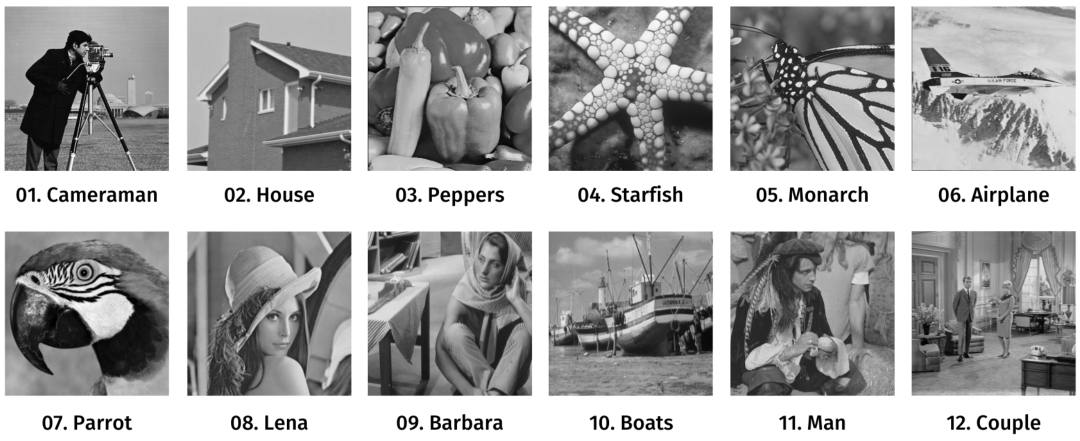

Figure 2.2. *Commonly used dataset to test image denoising algorithms called Set12 [23]. Includes images with both smooth textures and rather repetitive patterns.*

As the majority of image denoising papers use PSNR to compare an algorithm performance, we will also focus on using this measure to provide proper analysis. However, there are more quantitative characteristics to measure an algorithm's performance, e.g. Structural Similarity (SSIM) [31].

Now, when the denoising problem is introduced and the quantitative characteristic to compare different methods is established, let's describe the algorithms and techniques starting from the oldest to the most recent ones.

2.2 Not learning-based methods

Algorithms in this section are widely used in industry and considered classical and time tested. We start this chapter with the general overview of the not learning-based image denoising methods. In the later chapters, we will employ some of the non-learning based algorithms to seek improvement in learning-based methods.

First, we will consider Non-local Means algorithm and then proceed to its more robust extension Block-matching and 3D Filtering and Weighted Nuclear Norm Minimization.

2.2.1 Non-local means (NLM)

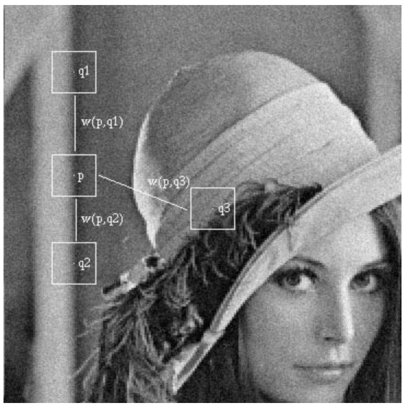

Figure 2.3. *Scheme of NL-means strategy. Similar pixel neighborhoods give a large weight, $w(p,q1)$ and $w(p,q2)$, while much different neighborhoods give a small weight $w(p,q3)$. [6]*

NLM algorithm was introduced in 2005 by Buades et al. [6]. The algorithm exploits self-similarities within an image to perform denoising. The underlying idea of the NLM algorithm that a natural image contains repeated patches inside itself and those similar patterns can be employed to estimate denoising procedure to a given patch. For a given square window reference patch of the image, its estimate is calculated using a linear combination of all other patches that the image contains. An example is provided in Figure 2.3 where for the reference patch p a linear combination of three other patches $q1$, $q2$, and $q3$ taken with corresponding weights $w(p,q1)$, $w(p,q2)$, and $w(p,q3)$. In practice, non-local means algorithm considers patches only in some bounded neighborhood of the reference patch to reduce the computational complexity. To find coefficients of the linear combination, the Euclidean distance between the reference and the current patches is used. Patches with a higher similarity have a bigger weight value and therefore bigger impact on the estimated patch.

Mathematically speaking, utilizing the algorithm for a noisy image $V = \{V(i), i \in I\}$, each pixel i in an image I can be estimated using a linear combination of all the other image pixels by applying the following formula:

$$NLM[V(i)] = \sum_{j \in I} \omega(i,j) \cdot V(j),$$

where weights selection based on the similarity of the pixel i and j and have the following properties:

$$0 \leq \omega(i,j) \leq 1, \sum_{j} \omega(i,j) = 1.$$

Weights themselves can be calculated as:

$$w(i,j) = \frac{\exp\left(-\frac{d(i,j)^2}{2l^2}\right)}{\sum_j \exp\left(-\frac{d(i,j)^2}{2l^2}\right)},$$

where Gaussian function is in the numerator and denominator does normalization to guarantee the weight properties. l is a filtering parameter depending on the noise standard deviation σ.

Euclidean distance (and weighted Euclidean distance):

$$d(i,j) = \int_{R^2} (V(i+t) - V(j+t))^2 \cdot G_a(t)dt,$$

where Gaussian window function $G_a(t)$ can be changed to the box function. a is a standard deviation of the Gaussian kernel.

From Figure 2.3 it is seen that patches $q1$ and $q2$ contain similar patterns as the reference patch p, therefore they have strong similarity and their weights will be higher than the weight of patch $q3$ that is taken from the different area where the pattern is different. The idea of utilization self-similarities of the images is laying in the NLM method. The denoising principle based on the collaborative filtering of the non-local patches called Non-local Filter (NLF). This method formed the basis of other more efficient methods we will discuss below.

2.2.2 Block-matching and 3D filtering (BM3D)

BM3D algorithm [8] is more sophisticated in comparison to NLM we described in the previous subsection. BM3D incorporates the principles of NLM but is not limited to them. It was developed in 2006 by Dabov et al. soon after NLM. Note that the algorithm was proposed at the Tampere University of Technology. In the following years, BM3D algorithm served as the basis for further development and lead its authors to a series of articles [9, 10, 11].

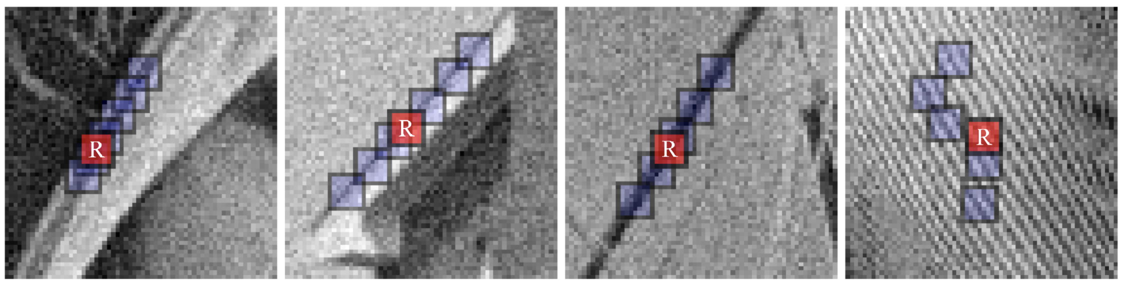

Figure 2.4. *Fragments of Lena, House, Boats and Barbara [Fig. 2.2] corrupted by AWGN of* $\sigma = 15$. *For each fragment block-matching is illustrated by showing a reference block marked with 'R' and a few of its matched ones. [8]*

At the first step, the algorithm is looking for mutually correlated patches to the reference one using the sliding window on the image corrupted by AWGN. The similarity between patches is measured by calculating the Euclidean distance, considering correlated patches up to the appropriate threshold value distance. Most correlated blocks (colored in blue) to the reference patch R (colored in red) depicted in Figure 2.4 for clarity. This phase of the algorithm called block-matching. Then 3D filtering part of the algorithm takes place. For each reference block, the correspondent patches stacked together in a so-called 3D cube ordering by descending distances with respect to the reference patch. The blocks in a 3D cube are very much correlated. To decorrelate the patches and obtain a sparse representation of the signal, three-dimensional decomposition is applied. Then the sparse representation of the noisy signal is filtered with, for example, Wiener filter where transform coefficients are filtered using thresholding. This phase of the algorithm called collaborative filtering. After filtering, the 3D signal is transformed back using the inverse transform, following by an NLM-like algorithm. On that last step, the patches are added as a linear combination similar to the NLM denoising strategy. Though, unlike in the NLM algorithm where only the central pixel is considered for estimation purposes, BM3D utilizes all the pixels in the 3D cube to reconstruct the image block from the noisy observation.

BM3D has been developed quite a long time ago, however, it is still one of the most notable and proven image denoising algorithms. It is used as an industry benchmark and implemented on various devices. The performance of newly developed image denoising algorithms are always compared to BM3D by default. The importance of this algorithm cannot be ignored and we will see that later in this thesis work when discussing a combination of learning-based approaches for image denoising along with BM3D iteratively.

2.2.3 Weighted Nuclear Norm Minimization (WNNM)

WNNM method for image denoising was introduced more recently in 2014 by Lei Zhang et al. [15]. The performance of WNNM is slightly better comparing to BM3D algorithm we discussed in the previous subsection. As in the BM3D, it is based on actively using non-local self-similarities within the image. The method utilizes low-rank matrix approximation methods, where different weights are designated to different singular values to minimize the nuclear norm, which is calculated as the sum of the singular values.

First, like in the above-mentioned problems, the algorithm explores and groups similar patches to the given reference one using, for example, block matching technique [9]. If we denote by Y_j a 3D matrix of stacked together similar noisy image windows (where Y is predefined fixed window size and j is window index position), we lead to the well-known equation of type

$$Y_j = X_j + N_j$$

where X_j is a patch matrix of clean image and N_j is noise itself. Matrix X_j is correlated with the matrix Y_j and apparently, X_j has a low rank being a vector of stacked non-local similar patches. Using approximation methods matrix X_j can be estimated from matrix Y_j. The same procedure that is described above for a single patch can be done for every patch, that the image comprises to then obtain the estimate of the whole image.

To estimate X_j the energy function is used:

$$\hat{X_j} = argmin_{X_j} \frac{1}{\sigma_n^2} \|Y_j - X_j\|_F^2 + \|X_j\|_{w,*}$$

adopting the known noise variance σ_n^2 to normalize the F-norm data fidelity term $\|Y_j - X_j\|_F^2$, where $\|X_j\|_{w,*}$ is the weighted nuclear norm and defines as a sum of its weighted singular values

$$\|X_j\|_{w,*} = \sum_i |w_i \sigma_i(X)|_1$$

where weights $w = [w_1, ..., w_n]$ and $w_i \geq 0$ is weight assigned to $\sigma_i(X_j)$, the i-th singular value of X_j, inversely proportional to singular value $\sigma_i(X_j)$ following the idea of natural images that the larger the singular value, the more its significance, and the less it should be influenced by the weight. Taking into account that the noise is distributed equally in every subspace spanned by the basis U and V of Singular Value Decomposition (SVD) $[U, \Sigma V] = SVD(Y_j)$, $\sigma_i(X_j)$ can be estimated as

$$\hat{\sigma_i}(X_j) = \sqrt{max(\sigma_i^2(Y_j) - n\sigma_n^2, 0)}$$

where $\sigma_i^2(Y_j)$ is i-th singular value of Y_j. The estimation is then

$$\hat{X_j} = US_W(\Sigma)V^T$$

where $S_W(\Sigma)$ is the generalized soft-thresholding operator with weight vector w,

$$S_W(\Sigma)_{ii} = max(\Sigma_{ii} - \omega_i, 0)$$

and $Y = U\Sigma V^T$ is the SVD of Y.

Shortly, nonlocal similarities are explored to group similar patches, weights are estimated as the inversely proportional to singular value vector, SVD is done for the stacked noisy patch to then get an estimation of the patch. Applying instructions above to each patch and then combining them together will lead us to the reconstructed image estimate. The procedure can be applied iteratively to obtain even better results of image restoration.

2.3 Discriminative learning-based methods

So far, we became familiar with the purpose of image denoising and the algorithmic not learning-based solutions for the denoising process. Now we will consider different approaches to this matter. Discriminative methods aim to distinguish the noise patterns based on learning to produce accurate estimates of the images.

The learning process takes a lot of memory and computing power. That is why this approach has been applied only in recent years. During the model training, many similar computations are performed consequently that can be running in parallel. With that, most of the training happens on the graphics processing unit (GPU) as it is more efficient than on the central processing unit (CPU). Notwithstanding, learning-based methods, in general, were known for more than half a century when theories about brain mechanisms began to evolve starting with a computer model of perceptron and a Multilayer Perceptron (MLP) to simulate the ability of the brain to recognize and discriminate. In addition to high computing capacity, these methods require to have large publicly available data sets that are made available to the wide public in recent years.

In this section, we will first understand the general image denoising pipeline for discriminative learning. Then we will overview the most relevant state-of-the-art networks. In particular, those networks that allow comprehensive analysis to be done in terms of hyperparameter selection and input transformation adoption will be finetuned. The following algorithm list is motivated by the overview of image denoising algorithms based on deep learning [20] that includes convolutional neural networks and wavelet neural networks.

We assume that the reader is familiar with the basics of the Convolutional Neural Networks (CNNs) and their elements, so that we can focus on the network architectures and parameter analysis, rather than dedicate an entire section to explain what is CNN and what it comprised of. Nevertheless, the necessary material and reasoning are given simultaneously along the way to fulfill the research.

The CNNs studied in this section for image denoising, in theory, can be replaced by fully connected neural networks. In [3, 26] reasoned that CNNs and neural networks with fully connected layers can be used interchangeably, and indistinguishable results can be achieved. However, the use of fully connected layers requires dramatically more computer resources and time since they demand the number of weights corresponding to the input features to be processed. Hence, we will acknowledge CNNs in the following sections.

2.3.1 Denoising pipeline for learning-based methods

Before moving to the actual learning-based methods, we need to understand the general differences between them and not learning-based approaches we discussed in the previous section. If we assume for now that the denoising algorithm itself is a black box, then a contaminated image is given to the black box in order to obtain an estimate of the image. This looks like there is no inconsistency with the not learning-based pipeline yet. However, there is a meaningful change since this black box needs somehow to learn and this occurs when its output is mapped to the clean undisrupted image we call ground truth. Hence, in the learning-based methodology, the ground truth image used twice during the process. Once as before, to calculate the PSNR value as a measure of black box robustness in image denoising operation. And the other time ground truth image is used when given to the black box so that the network is capable to learn what the resulting image should look like. This change requires having a sufficient set of training examples for the learning process. The size of the set can be though easily increased with augmentation techniques applied to the training set to expand it.

Providing examples of clean ground truth images to map them to input noisy ones requires to define a measure of how successful the network performs on each iteration. This is done with objective function, for example, MSE, we already mentioned before in Section 2.1. During the learning process the weights of a network are learned and adjusted using the objective, or loss function. When the loss is acceptably small and does not drop anymore, we assume the training process is saturated, and the learned weights can be used to test the network. When testing, the images that network has never seen during the training process are provided for the fair and unbiased evaluation. In testing the images are still distinguished as ground truth and corrupted image, as we need quantitative characteristics to compare the network output estimate result with the desired clean image, using in our case PSNR. The learning process can be accelerated by providing a batch of multiple images as the input to average the error for several images, as a single image does not have much effect on the learning process.

In the learning-based pipeline, it is necessary to pay attention to possible variations in the output of the black box. The output of the black box can be an estimate of a clean image as aforementioned. In this case, the algorithm learns how the real image looks like when mapping the estimate result of the black box to the ground truth. Instead of learning the images that can be very different from one to another, it is more natural to learn the noise model, which stays the same for all the images in our experimental set (AWGN in our settings). When we accept that the black box should learn and output the noise, as a mapping we need to provide the noise itself that we added to the clean ground truth image before feeding to the black-box algorithm. The noise-learning process could be modified with an additional step inside the black box, so the algorithm learns the noise model, though the output of the black box is an image estimate that can be mapped then to the ground truth image, not to the added noise. To do that, the last step of the black box should output a subtraction of the noisy input image and the learned noise estimation. This step can be incorporated into the black box as it is a predefined linear operation and does not affect the learning process of the black box. With this step, the output again mapped to the ground truth, though the noise model is learned in this case, not the image model. The above-mentioned pipeline for noise estimation is commonly used in learning-based denoising methods we will be dealing with in the further subsections and called residual learning.

2.3.2 Denoising Convolutional Neural Network (DnCNN)

DnCNN was developed recently by Zhang et al. in 2016 [32] and depicted in Figure 2.5. DnCNN is simple by its structure to perform comprehensive analysis, yet very efficient deep CNN, components of which will be studied, tuned and evaluated in this thesis.

Deep network is operating with noisy observation described by the formula we already used before: $y = x + n$, where y is noisy observation acquired as a sum of clean image x and degraded by AWGN n. DnCNN uses residual learning to build an estimate of the ground truth image $\hat{x}$. During the training process, the network learns the residual mapping $\mathcal{R}(y) \approx n$. The last step inside the network is linear and therefore non-trainable: the learned noise model subtracted from the noisy observation to form an estimate, $y - \mathcal{R}(y) = \hat{x}$. Instead of using input noisy image and image estimate to calculate the loss, the average Mean Squared Error between the estimated noise and the real noise (residual image) is adopted to support the training process:

$$L(\Theta) = \frac{1}{2N} \sum_{i=1}^{N} \|\mathcal{R}(y_i; \Theta) - (y_i - x_i)\|_F^2$$

where $\{(y_i, x_i)\}_{i=1}^{N}$ are N pairs of noisy and the corresponding clean images (or patches) for DnCNN training.

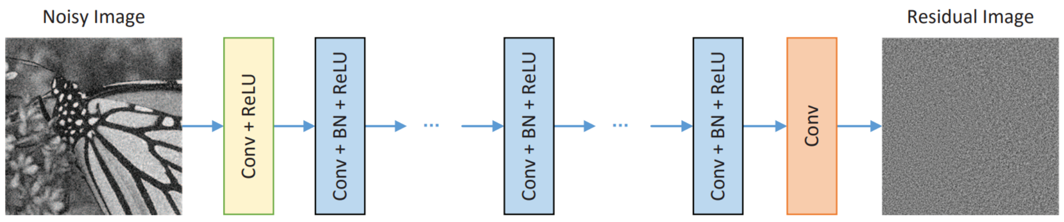

Figure 2.5. *General architecture of DnCNN network [32].*

The basic DnCNN suggested in [32] consists of 17 convolutional layers for grayscale images, 15 of which are hidden (colored in blue in Figure 2.5). Although we do not consider operating with color images, it is worth mentioning, that for color images (in particular, 3-channel RGB images) already 20 convolutional layers are employed to reach the best performance in DnCNN [32]. In the first layer (Conv + ReLU) 64 convolutional kernels of size 3 by 3 are used to extract corresponding number of features from an input noisy grayscale image, followed by Rectified Linear Unit (ReLU) activation function. Batch normalization (BN) is done before ReLU in each hidden layer (Conv + BN + ReLU), where the other parameters stay the same. In the last layer, only one convolutional filter 3x3 is utilized in order to obtain the residual image.

The pipeline of the training process for DnCNN is the following: first, a grayscale image x is corrupted with AWGN model n with standard deviation $\sigma = 25$. To simplify the process description, we say an image; however, we are dealing with a batch of images. In the next step, the noisy image input y is passing through the network. In the end, the noisy grayscale input image y is subtracted with the output of the network $\mathcal{R}(y)$ to provide an image estimate $\hat{x}$. However, as said above, the estimate of the noise $\mathcal{R}(y)$ used to calculate the loss (not the image extimate $\hat{x}$), as average MSE between estimated noise $\mathcal{R}(y)$ and the desired residual image $y - x = n$. With the subtraction step at the end of the network flow, the neural network learns the noise model (though, the output is an estimate) and is called a residual network. After training, the obtained model tested on twelve images widely used in digital signal processing, comprising a dataset, called Set12, and illustrated in Figure 2.2, to calculate PSNR value.

Note that two networks are considered in [32]: one trained with the fixed noise level, called DnCNN-S for specific noise, and the other one trained with images corrupted with altering noise level $\sigma \in [0, 55]$, called DnCNN-B for blind noise model training. In this work we consider training and testing using specific noise model ($\sigma = 25$), where sigma fixed during the training, unless otherwise specified.

2.3.3 FlashLight CNN (FLCNN)

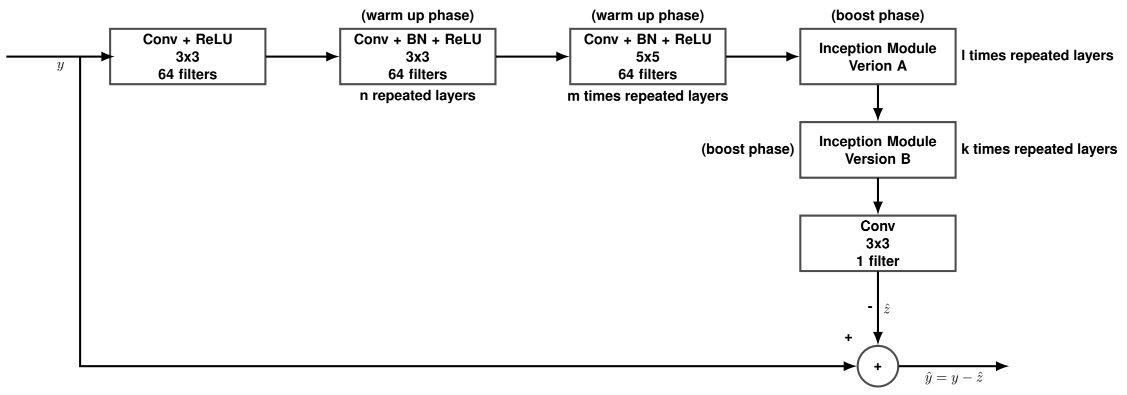

Figure 2.6. *FlashLight CNN (FLCNN) architecture for denoising. The recovered images (estimates) $\hat{y}$ are constructed from noisy images (y) and predicted noisy measurement ($\hat{z}$) [5]*

The other network considered in this chapter is called FlashLight CNN (FLCNN). It is far more complex to analyze comparing to DnCNN. FLCNN was proposed and studied in thesis work [5] and later in the article [29]. Its architecture was inspired by both DnCNN [32] and inception network [27] and depicted in Figure 2.6. FLCNN is composed of two cascaded phases: the consecutive convolutional DnCNN-like layers followed by the phase which employed the residual inception layers in different combinations. The model utilizing the inception layers with identity shortcut connections in its so-called inception modules (Figure 2.7 and 2.8) is shown in the general scheme (Figure 2.6) as Inception Module Version A and Inception Module Version B. Similarly to DnCNN, the network is trained on the images corrupted by the same fixed noise model (with standard deviation $\sigma = 25$). Mean PSNR value for FLCNN tested on Set12 is 30.59 [5], which is about 0.2dB higher comparing to DnCNN model, making this network state-of-the-art at the time of its publishing.

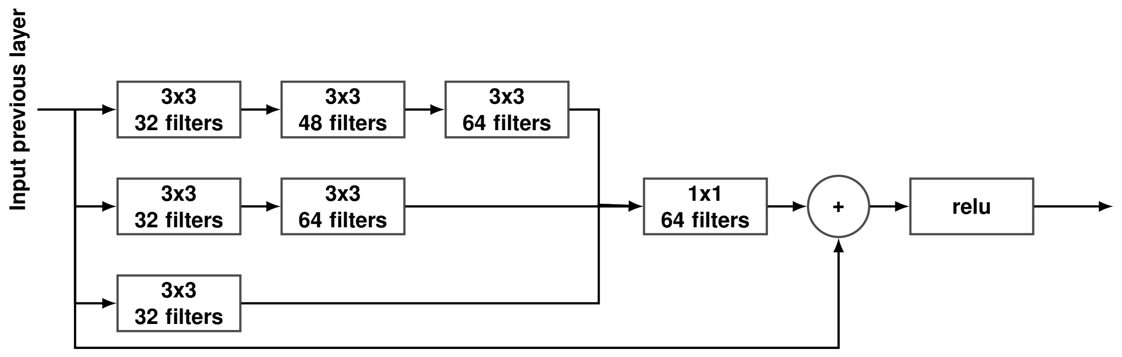

Figure 2.7. *Inception layer version A [5]*

FLCNN comprises several concepts at once, such as inception model, identity connections, utilizes receptive field ideas and also includes DnCNN-like convolutional layers with different kernel sizes. As in DnCNN, the first layer has (Conv + ReLU) 64 convolutional kernels of size 3 by 3 are used to extract 64 features from the input noisy grayscale image, followed by Batch normalization and ReLU activation. Then, similar to DnCNN layers (Conv + BN + ReLU) are used. FLCNN has 5 such layers with 3x3 kernels, which are part of the so-called warming up phase. Next, 4 identical layers with bigger kernels 5x5 wrapping up the warming up phase. After the DnCNN-like layers, two similar inception modules are used. Inception module version A layer incorporates three parallel branches and identity connection, shown in Figure 2.7. The outputs of the parallel branches are then concatenated and processed with 64 kernels of 1x1 to extract 64 linear combinations for each pixel.

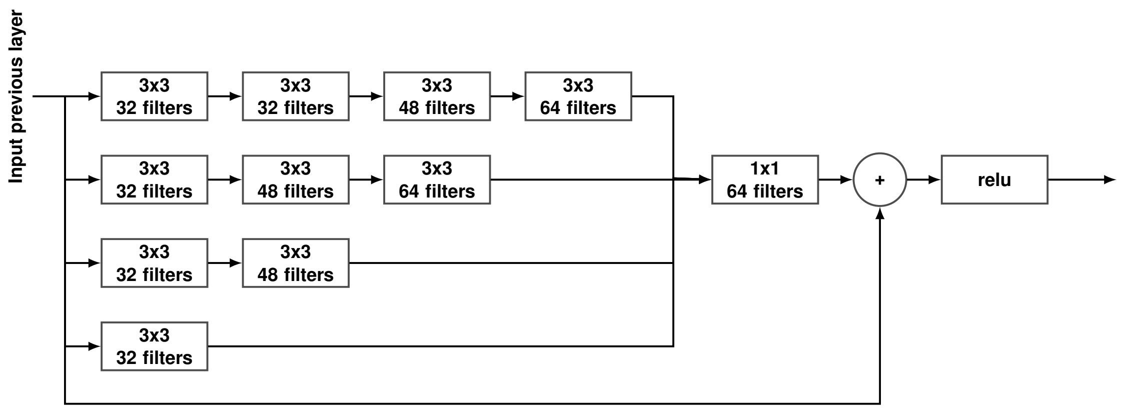

Figure 2.8. *Inception layer version B [5]*

Kernels in parallel branches have all the same size 3x3, but represents the different kernel size in terms of receptive filed as it will be discussed in analysis part of the thesis in Section 3.1.5. Two consecutive 3x3 kernels act similarly as a 5x5 kernel, require fewer computations and bring more effective feature extraction. Similarly, three consecutive 3x3 kernels represent 7x7 kernel. Inception module version B is similar to version A, it includes four parallel branches with one more level of feature extraction using four consecutive 3x3 kernels, roughly representing the receptive field of 9x9 kernel. FLCNN consists of 14 layers (n=5, m=4, l=3, k=2 in Figure 2.6). The concepts we mentioned in this subsection will be thoroughly discussed one by one in Section 3.1 where these techniques are analyzed and then numerically tested.

2.3.4 Fast and Flexible Denoising CNN (FFDNet)

Discriminative learning methods for image denoising evolved rapidly and already in 2017 Kai Zhang et al. submitted a new deep network architecture, fast and flexible denoising CNN they called FFDNet [33], just one year after their team introduced already familiar DnCNN. While the network architecture is something similar to DnCNN with convolutional layers stacked one after another (illustrated in Figure 2.9), there is a number of important differences with the most notable one in the input and output entities of the network. FFDNet works with the downsampled sub-images rather than with spacial images, so the input is not plain, but multidimensional. Multi-layer inputs like that we will also call cubes or 3D cubes.

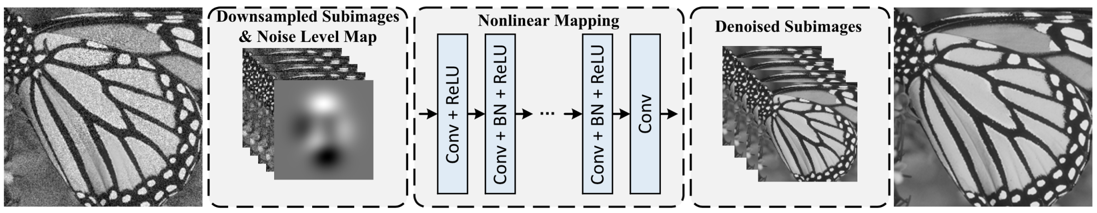

Figure 2.9. *General architecture of FFDNet network [33].*

A spatial grayscale image is downsampled in a certain way to form the basic part of the input before it passed further to the non-linear mapping part of the network. The pixels of the initial image are rearranged by the network so that for each 2×2 non-overlapping block all four pixels of this patch will be placed in a separate channel, as shown with colors in Figure 2.10. For an input image with dimensions $n_{ch} \times h \times \omega$, the subsampled image will be a 3D cube of size $4n_{ch} \times h/2 \times \omega/2$, where n_{ch} is the number of channels, h and ω are height and width correspondingly. In the simplest grayscale case with a spatial image ($n_{ch} = 1$), the subsampling procedure will lead us to a 3D cube with four channels. In addition, the noise level map is stacked to this cube as a fifth channel to finally settle the network's input. The network receives and further processing on this 3D cube representation of the image.

Most of the discriminative learning methods work well only for specific noise level and you will need to train separate models for different noise models, while the proposed FFDNet has a tunable noise level map as one of the input layers to tackle the problem of multiple models depending on the noise. In this sense, at first glance, the network could have been treated as blind since the network is trained once and the obtained model stays the same for any noise, though it would be incorrect. To recall, we discussed blind networks

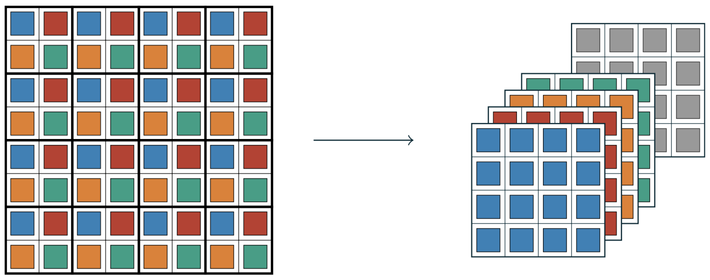

Figure 2.10. *FFDNet downsampling [28].*

in subsection 2.3.2 when going through the blind version of DnCNN, namely DnCNN-B that is trained on images corrupted with different noise models so that it used to recover an image without prior knowledge or noise estimation). It is crucial to understand that the noise level is provided or estimated in advance and passed to the FFDNet network as one of the input layers. In our study case with AWGN use, the tunable noise level map is a plane with uniformly distributed sigma values. However, in more non-trivial cases with unevenly distributed noise (spatially variant noise), this layer contains a non-uniform noise level map.

The basic FFDNet [33] has 15 convolutional layers, among which 13 are hidden (Figure 2.9). Same as in DnCNN, in the first layer (Conv + ReLU) 64 convolutional kernels of size 3 by 3 are used to extract corresponding number of features from 5-channel input cube, followed by Rectified Linear Unit (ReLU) activation function. Batch normalization is done before ReLU in each hidden layer (Conv + BN + ReLU), where the other parameters stay the same. In contrast, in the last layer, 4 convolutional filters 3x3 are utilized in order to obtain the denoised sub-images. In the end, the inverse transform is done to assemble the spatial image estimate from the 3D cube output of the network.

The output of the FFDNet network has a smaller dimension than the provided input, meaning that the subtraction between the output and the input on the last step inside the network is not possible like it was done in DnCNN. Opposite to DnCNN, the output of the FFDNet network (after being assembled back from the cube to a spatial image) is not a residual image, but an image estimate itself. That means the network learns the images, not the noise model.

The estimate obtained with FFDNet can be formalized as $x = \mathcal{F}(y, \mathbf{M}, \Theta)$, where $\mathbf{M}$ is a noise level map, y is a noisy input image, Θ is the model parameters. When training, the loss between the image estimates and the ground truth images computed and minimized using the formula:

$$\mathcal{L}(\Theta) = \frac{1}{2N} \sum_{i=1}^{N} \|\mathcal{F}(y_i, \mathbf{M}_i; \Theta) - x_i\|^2$$

The network obtains the noisy images by corrupting grayscale ground truth images with AWGN model with standard deviation $\sigma \in [0, 75]$. With that FFDNet covers wide range of noise levels with a single trained model.

In the end, when FFDNet tested on Set12 with different noise level σ-values, it appeared that it gives better results for most of the images comparing to BM3D, WNNM and DnCNN. FFDNet performs particularly good on the images corrupted with noise model with $\sigma = 25, 50, 75$ - better than other networks mentioned above for almost all images from Set12 with minor exceptions.

2.3.5 Wavelet CNN (WCNN)

The new results in image denoising using state-of-the-art deep learning approaches appeared in the same year. In 2017 Korean team introduced a deep residual network for image restoration problems [4] that we will refer to as Wavelet CNN (WCNN) in this thesis. One of the main goals of the work was to overperform signal processing approaches such as BM3D on the images that have many repetitive patterns, where CNN-based methods are still inferior.

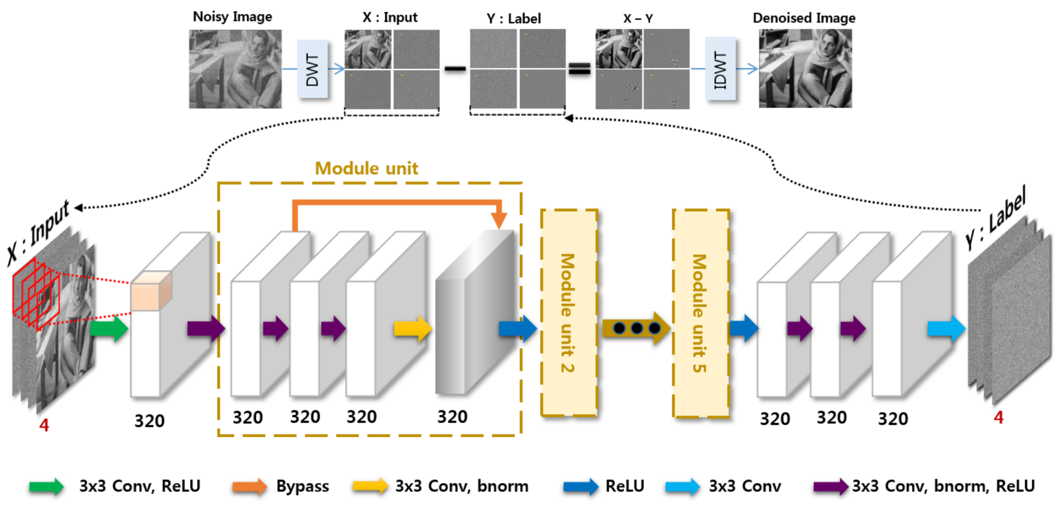

Figure 2.11. *General architecture of Wavelet CNN (WCNN) [4].*

WCNN is the first network in this thesis that utilizes Discrete Wavelet Transform (DWT) with one-level Haar wavelets (Haar DWT, or HDWT) to convert an input image to a sparsified 3D cube wavelet representation by passing the signal through a series of filters and then stacking the filtered results as shown in Figure 2.11 namely X. For example, a grayscale input image of size $n \times n$ transforms into a four-channel data cube with shape $n/2 \times n/2 \times 4$. Four downscaled channels yield to the four filters: LL, HL, LH, and HH, where capital L stands for a low-pass filter, capital H is for a high-pass filter. Two letters indicating that grayscale images are 2D signals and filtered in two dimensions by sequentially applying DWT-transformation both vertically and horizontally. LL subband looks like a smaller copy of the image as it obtained with low-pass filters, while the other three preserve only image details, such as edges. HL subband keeps horizontal details, LH - vertical, and finally, HH represents diagonal features. When the transform has completed, four fragments (subbands) stacked together to form the network's 3D cube input.

The output of this residual model has the same shape as the input, where each channel contains the corresponding noise in the wavelet domain. These four wavelet subbands for noise estimate then reshaped from a 3D cube back to the spatial wavelet representation with four wavelet quadrants. Further, the DWT of the noisy input subtracted with the noise estimate in DWT domain to obtain an estimate, denoised image in DWT domain. Finally, Inverse Discrete Wavelet Transform (IDWT) performed on the spatial wavelet image estimate quadrants to provide the estimated image in the spatial domain.

WCNN have slightly more complex structure comparing to, for example, beforementioned DnCNN and FFDNet networks. First, the network manipulates with input in the wavelet domain. Inside its hidden layers, it has five equal modules. Each module consists of 3 convolutional layers (Conv + BN + ReLU). Each convolution extracts an immense number of features, specifically 320 features in each convolutional layer using 3x3 kernels. Second, as a complication, each module has an identity connection from the first convolutional layer inside the module to the last one, as shown in Figure 2.11. Overall WCNN contains 20 convolutional layers. Although the kernels used are small (3x3), with the exhaustive number of features extracted with these kernels, this makes this network computationally heavy.

2.3.6 Multi-level Wavelet CNN (MWCNN)

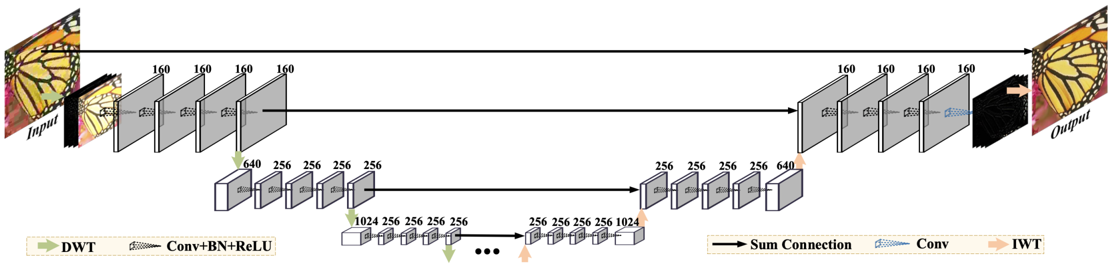

Figure 2.12. *General architecture of MWCNN network [21].*

In the following year, in 2018, a Chinese team that also included authors of DnCNN and FFDNet introduced a Multi-level Wavelet-CNN they called MWCNN [21] to tackle image restoration problems. In the MWCNN paper, a novel model for a better tradeoff between receptive field size and computational efficiency is presented. The architecture of MWCNN was inspired by the U-net network [25], which was introduced for biomedical image segmentation purposes by a German team much earlier, in 2015.

MWCNN general architecture is depicted in Figure 2.12. As WCNN, it utilizes DWT, however, not only for the input image but also for the features inside the network layers. Instead of pooling layers used in U-net to downsample the size of the feature map in the hidden layers, MWCNN replaces them with DWT that keeps the information, both in frequency and localization.

The idea of multi-level DWT utilization is depicted in Figure 2.13. On the first level, a

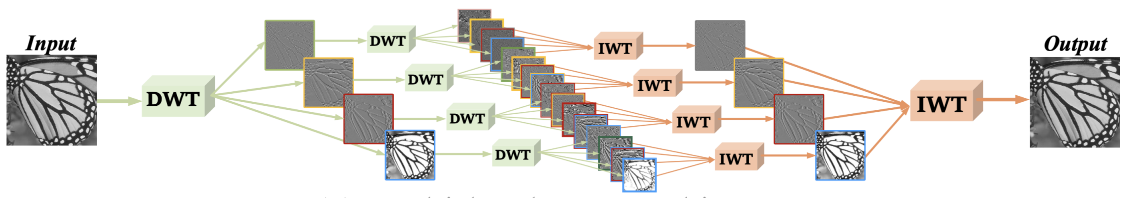

Figure 2.13. *Multi-level wavelet transform architecture [21].*

grayscale input image decomposed with a 2D discrete wavelet transform on four subbands that we stack together into what we called a wavelet cube. On the second level of wavelet decomposition, each of the four subbands goes further through the DWT, making the depth of the wavelet cube grows four times, from four to sixteen. This procedure can be applied to further levels. Each packet of four cube layers (LL, HL, LH, and HH bands we first mentioned in Section 2.3.5) corresponds to a wavelet transform and can be accurately reconstructed back by the IDWT. With that one to one reconstruction, the output image is identical to the input in Figure 2.13 after the last IDWT is employed.

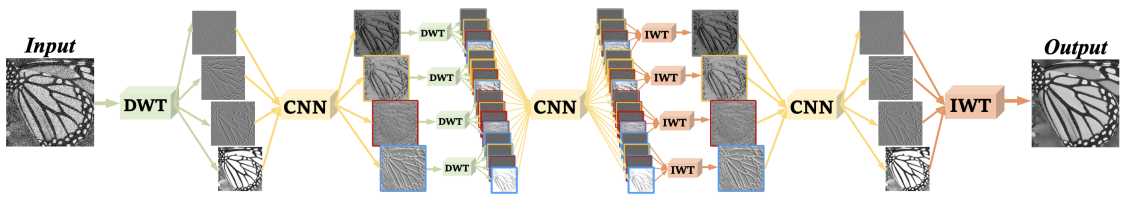

Figure 2.14. *Multi-level wavelet-CNN architecture [21]*

Expanding the multi-level DWT idea with utilization of non-linear mappings achieved with convolutional layers between any two levels of DWT is shown in Figure 2.14. Determining the CNN blocks between DWT layers and adding identity connections brings us to MWCNN architecture shown in Figure 2.12.

The MWCNN network consists of two symmetric parts, contracting and expanding subnetworks. Both contracting and expanding subnetworks have similar blocks on different levels that are connected with the identity connection layer. In the contracting subnetwork, the input image (say the input grayscale image size is $N \times N \times 1$) first downsampled from a spatial domain into a wavelet domain (using DWT with one-level Haar wavelets). Also, the identity connection from the spatial input image to the output on this level has been established (called Sum Connection in Figure 2.12). Four bands of the input's wavelet transform stacked together to form a wavelet cube of the size $N/2 \times N/2 \times 4$. The first level of the contracting network contains a block of four convolutional layers (Conv + BN + ReLU), where each convolution extracts and propagates further 160 features, preserving the size of the wavelet cube. The filter kernel size in all convolutions is 3×3. In the next level of MWCNN, the feature map is downsampled with DWT, where wavelet transform replaces the pooling operation. Invertibility of DWT assures that all the information is preserved when the wavelet transform downsampling operation replacement is done. Each of the $N/2 \times N/2 \times 160$ feature maps is passed through the DWT transform. All of the DWT bands are stacked together, forming a wavelet cube with a twice lower resolution and four times more of the feature maps ($N/4 \times N/4 \times 640$). Then in this and the other layers of MWCNN follows a block of four convolutional layers (Conv + BN + ReLU) extracting 256 features in each layer. In the expanding part of the network, similar processes happen, and an intermediate wavelet cube image is upsampled with inverse transforms. Before each block of four convolutional layers starts, the feature map is obtained from the higher level with IDWT performed on the wavelet cube. To do the IDWT, the wavelet cube needs to be disassembled the same way it was formed in the contracting network so that the respective layers (LL, HL, LH, and HH bands) are taken as part of one wavelet plane. In the last convolutional layer, the convolution operation is used alone (without BN, and ReLU) to extract four features, or four wavelet bands, that corresponds to the residual image in the wavelet domain (black layer on the right in Figure 2.12). The residual image reconstructed with IDWT and contains the noise model learned by the network. Using the identity connection, the input noisy image then added to the predicted noise model to output an estimate for the clean image. The noise prediction learned by the network actually needs to contain values that are opposite to the added noise (or the subtraction need to be performed for the identity connection) as the input noisy image is added to the predicted noise model in the scheme. With the identity connection, the residual learning formulation is incorporated in MWCNN.

The network output can be written mathematically as $\tilde{x} = \mathcal{F}(y, \Theta)$, where Θ is the network parameters, y is a noisy input image. When training, the loss between the image estimates $\tilde{x}_i$ and the ground truth images x_i computed with the objective function:

$$\mathcal{L}(\Theta) = \frac{1}{2N} \sum_{i=1}^{N} \|\mathcal{F}(y_i, \Theta) - x_i\|_F^2 .$$

MWCNN tested with images from the Set12 [Fig. 2.2] corrupted with the noise level $\sigma = 25$ and slightly outperforms both non-learning methods, including BM3D and WNNM, and learning-based methods, particularly DnCNN and FFDNet in terms of PSNR.

2.4 Combined methods

In this section, we will discuss how non-learning based methods and discriminative based methods can be combined to leverage the image denoising problem. Both approaches alone have their advantages and drawbacks. NLFs we described above provide good quality denoising results notably on the images that represent strong self-similarity patterns. However, on a generically structured image, these algorithms still yield to CNN-based, that on the other hand are able to extract complex image features, but may give an inferior performance on repetitive textures. In addition, the training process requires, in most cases, a significant amount of time, up to several days, to train a network that may only work best for a specific noise type and level. Nevertheless, this is not the rule, as for example, reviewed earlier in Subsection 2.3.5 WCNN, that actually enhances DnCNN by employing DWT to its input, surpasses both WNNM and BM3D on the specific images from Set12 that have distinctly repetitive patterns (House, Starfish, Monarch, Lena, and Barbara images from Set12 [Fig. 2.2]).

Some of the architectures combine NLFs and CNNs. For example, Block-Matching Convolutional Neural Network (BMCNN) [2] using BM3D approach, but instead of Wiener filtering for similar patches collaborative processing, it uses non-linear mapping employing convolutions for denoising, similarly to DnCNN. However, the BMCNN still need to be retrained for images of the different noise levels.

In this section, we consider more closely the so-called plug-in approach that accepts any generic CNN-based filter (CNNF) with any generic NLF, without retraining necessity.

2.4.1 Nonlocality-Reinforced Convolutional Neural Netwroks (NN3D)

A paradigm for independent using of a combination of CNNF and NLF is described here under the name of NN3D [7]. Disadvantages of CNNF and NLF are listed at the beginning of this section. Interestingly, that where one approach is inferior, the other one is likely to work robustly. In other words, the drawbacks of NLF can be neglected by using CNNF after, and vice versa, as NN3D applies an iterative strategy. NN3D comprises of two supporting blocks, a block where a generic CNNF applies to its input, following by a block, where the result of CNNF block is further processing by plugging in a generic NLF. To leverage the denoising effect, this procedure is done consecutively for K iterations as shown in Figure 2.15.

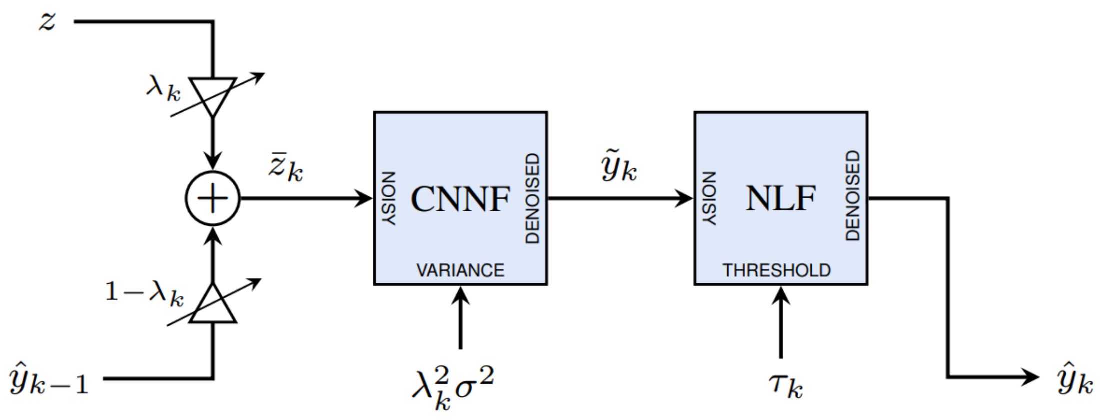

Figure 2.15. *General architecture of NN3D network [7].*

On the first iteration ($k = 1$), the noisy image z is provided as is as the input ($\bar{z}_1$) to CNNF block, then the filtered result $\tilde{y}_1$ is further passed as the input for NLF block. The output estimate after the first iteration is labeled as $\hat{y}_1$. On the next iterations, the initial noisy input each time blending in some proportion with an estimate attained from the previous step. The proportion of the initial noisy image varies each step and is denoted as λ_k, while the proportion of the estimate from the previous step taken with coefficient $(1-\lambda)$. λ_k is starting with 1 on the first iteration and decreasing with steps, staying positive. With every iteration, we acknowledge that every new noisy input $\overline{z}_k$ contains less noise. That requires CNNF to have pre-trained models for several noise levels, unless it is not blind. For example, if the initial image has been contaminated with the noise level of $\sigma = 30$, and we run NN3D for two iterations with $\lambda_{1,2} = [1; 0.5]$, then on the first iteration we will apply CNNF trained with $\sigma = 30$ for the input noisy image. At the second step, half of the initial noisy image will be combined with half of the estimated denoised image obtained after the first iteration. The noise level of the input on the second step $\overline{z}_2$ calculated as the multiplication $\lambda_2\sigma = 15$, and CNNF block trained with $\sigma = 15$ applied. On arbitrary step k, the noisy input $\overline{z}_k$ that feed to CNNF is calculated as $\overline{z}_k = \lambda_k z + (1-\lambda)\hat{y}_{k-1}$, with the noise level for this input $\lambda_k\sigma$.

When incorporating non-blind CNNs to the first block of NN3D, the network is not just required to know the noise level, we also need to have pre-trained models for all required noise levels the network is going to use. However, when FFDNet used in CNNF, it just required to know the sigma value, but it does the work, as we know, with only one trained model. If we do not have all the necessary pre-trained models, we can adapt an image, so that noise standard deviation matches with the available pre-trained models.

The advantage of NN3D is plug-in approach, so that we can easily switch between different CNNs in CNNF block, as well as change NLF method without retraining. However, if the selected CNNF is not blind, it has to be trained with multiple noise level to be used iteratively with decreasing noise. This approach provides numerous combinations to be plugged-in and gives excellent denoising results.

In the next chapters, we will go through some of the described networks, seeking better quantitative and qualitative results, concentrating particularly on PSNR value improvement. We will pay more attention to the structure and elements of the networks, introducing new concepts. Get acquainted with specific details of a network, such as network layers and image transformations. The reasoning will be supported by the experimental setups and their numerical results.

3 CNN FILTERING

There are many approaches for image denoising using different CNN architectures as we saw in the previous chapter. A network with a simple structure needs to be considered as a basic model to perform in-depth network component analysis. In this chapter, two efficient deep residual neural networks for image denoising, DnCNN and FLCNN are discussed and their parameter selection is examined. We mainly audit the parameters for DnCNN due to its plain structure. Parameters of the network are studied independently to evaluate their sole impact on the output results. We will seek network optimizations with every different configuration, also combining the parameters. First, we address the theoretical aspect of the techniques. Then in the later subsections, we come across the quantitative results. The analysis of the network covers approaches used in CNN development, such as shortcut connections and inception models. We will also discuss the effective receptive field concept, and why neural networks becoming deeper, but not wider. In this chapter, networks receive as an input a grayscale image in a spatial domain. There are no preparations or any transform done to the images before they proceed into a network structure. As a result of the parameter analysis, techniques that lead to better performance can be utilized in newly derived models.

3.1 DnCNN Parameter Analysis

The following concepts and parameters are applicable to any network. To simplify the analysis, we evaluate the network components based on DnCNN. In each of the following subsections, we first discuss the concepts in general and then apply them particularly to DnCNN.

3.1.1 Receptive field

The receptive field term itself comes from neural science, but its application is also found in the context of CNN. In a fully connected network, each neuron is connected to every single data input from its previous layer. More specifically, any intermediary layer pixel value is calculated as a linear combination of all previous layer pixel values. Unlike in fully connected networks, in CNNs neurons are only connected to a certain area of the foregoing layer. The input area that a single neuron can cover at a given time is called the local receptive field. Kernel size in a convolutional layer is defining the receptive field of this layer.

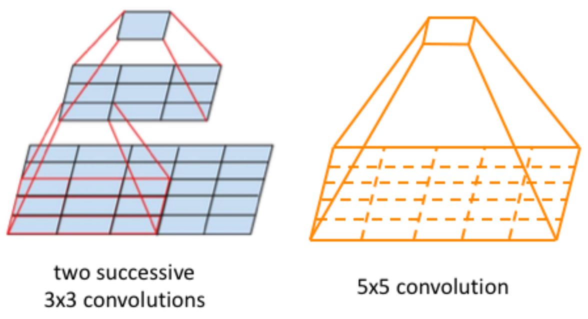

Figure 3.1. *Receptive field for different kernel sizes [18]. 5x5 receptive field by stacking two 3x3 convolutions.*

The illustration of the receptive field concept is provided in Figure 3.1. For example, when a 5x5 convolutional filter (orange-colored on the right side) slides across the input image, at each moment, it is exposed to a 5x5 area that is convolved with the kernel to produce one output value. The convolution result is obtained by multiplying 5x5 input block pixels from the receptive field with 25 corresponding kernel weights. In other words, the 25 pixels of the receptive field area are contributed to this one output pixel.

Another example of a 5x5 receptive field is depicted with blue color on the left in Figure 3.1. The same size receptive field in this case is constituted by two 3x3 convolutional kernels applied one after another (in two convolutional layers). So that the single pixel on the top is exposed to a 3x3 area in the previous layer, and its local receptive field is 3x3. But in their turn, each of these 3x3 values in the previous layer are also obtained by a 3x3 area, meaning that overall contribution to this layer made by 5x5 area. Note, that here we use stride=1, so that the convolutional kernel slides one by one without skipping pixels. Effectively, the top pixel value is generated not just by a 3x3 area in the previous layer, but also indirectly by the values in the earlier layer, that comprises much larger area. The effective receptive field is that part of the initial input image which pixels have been directly or indirectly contributed to the final output.

Sliding window mimics the human sight, where the center pixel contribute from the kernel contribute the most. As when you read, your eyes are moving to keep in focus what is in the center, while picture on the periphery is blurred. In practice, the effective receptive field size of the cascade convolutions is smaller than that of the single kernel [22].

Interestingly, we can use two 3x3 filters instead of one 5x5 kernel to have the same receptive field. That means the training process will operate with $2 \times 3 \times 3 = 18$ weights for two stacked 3x3 convolutions instead of $1 \times 5 \times 5 = 25$ weights for one 5x5 convolution. The fewer convolutional weights the network needs to learn, the better in terms of computations, as, in real life, the kernels have a third dimension and are represented by a tensor (meaning another multiplier in the weight calculation). For example, in DnCNN, the kernel size is $3 \times 3 \times 64$, as they operate on 64-channel intermediary image data. Also, real-life networks are deeper, and therefore the number of learning weights increases dramatically. In fact, a receptive field of any arbitrary size can be simulated with only 3x3 kernels by using more convolutional layers.

Now we utilize the receptive field ideas specifically to the DnCNN. DnCNN has $N = 17$ layers in total, having 15 hidden layers. Convolutional kernel size is $K \times K \times C$, where $K = 3$ and C is the number of channels of the convolutional input. For each network's output pixel value, its receptive field is calculated using the following formula:

$$RF = \big(K + (N-1)(K-1)\big) \times \big(K + (N-1)(K-1)\big). \tag{3.1}$$

Using the Equation 3.1 for DnCNN settings, the receptive field is $RF = 35 \times 35$ pixels. As we discussed above, the same size receptive field can be achieved by using fewer layers with larger convolutional kernels. But how the performance of the network change when the depth is reduced and the kernel support is increased?

In this section, we will study how a different number of layers and the kernel size (given the receptive field is constant) affects the actual performance measured in PSNR value. To preserve network's receptive field $RF = 35$ (hereafter we only use its one dimension), the relation between the kernel size K and the number of layers N is derived from Equation 3.1: $N = \frac{RF-1}{K-1}$, where $K, N \in \mathbb{N}$. As we observe, the relation is hyperbolic, and the shallower the network, the bigger the kernel size has to be. However, both the number of layers N and the kernel size K must be an integer, which slightly affects the receptive field size.

3.1.2 Network depth

In the image classification field, CNNs have become deeper in recent years. With that, in just four years the depth studied in the articles has been increased from 7 layers [19] to tens and hundreds of layers [16]. With more layers, we are able to extract more complex patterns at various levels of abstraction from the input.

As discussed in the previous subsection, the receptive field becomes wider with more layers, given fixed kernel size. We will set the number of experiments on DnCNN varying the depth of the network, and therefore varying the receptive field implicitly. We expect that the performance of DnCNN in terms of mean PSNR value increases with the use of more layers. Denoising ability increases notably only until the receptive field is smaller or equal to the training image size. The experimental results will be demonstrated below in the corresponding subsection.

3.1.3 Feature extraction

To find more complex relations in data, more features can be extracted in each layer. In these experiments, we fix the number of layers to the optimal value ($N = 17$, as we concluded from the numerical results for the network depth experiments). We will be changing only one parameter - the number of channels in the intermediary layers. Some of the networks [4] that extract many channels from the data show good performance for the immense number of features extracted in each layer. We expect that with more features extracted in each layers, more details from the initial input may be observed. However, a considerable number of features needs to be found, as extracting more features could bring redundant and dependent features.

3.1.4 Identity shortcut connections

In subsection 2.3.1 we mentioned for the first time the residual learning term when a network learns the noise model to remove the contamination from an image. One of the implementations for the residual learning was discussed during our familiarization with DnCNN in subsection 2.3.2, where we either map the output of the network to the noise itself or inject a subtraction step into the network to map the output to the ground truth image which is more natural. However, the problem of vanishing/exploding gradients arises when stacking more layers in the network. In [16] it has been shown that with more layers, the training error gets larger, while the network converges similarly to one with fewer layers. When testing, the error for the network with more layers is larger too, meaning that having just more layers does not bring learning capacity to the network. In this subsection, we discuss another technique for residual learning.

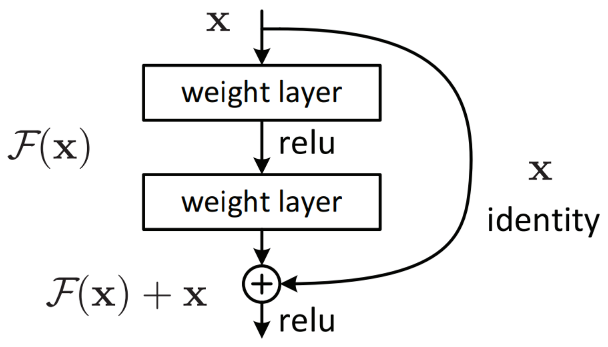

Figure 3.2. *Residual learning: a building block [16].*

Identity shortcut, or shortcut connections, or bypass, depicted in Figure 3.2 used in [16] to prevent degradation problem.

The idea behind this procedure is that most of the image features can be passed from one layer further as is, while only the noise part goes through non-linear transformations. Suppose for a moment a case where the input image does not have any noise to remove, the identity mapping propagating forward the image itself, and non-linear mappings operating with zero residual noise. It is easier to push the residual to zero than to fit an identity mapping by a stack of non-linear layers [16]. In Figure 3.2, the input we receive from the previous layer is denoted as x, same x we pass several layers further as identity. We also pass x to the convolutional layers that perform non-linear mapping $\mathcal{F}(x)$. Then the identity connection and the non-linear mapping are summed $\mathcal{F}(x) + x$ to form one layer.

Residual approach with the use of identity shortcut connections allows having more layers without a drop in performance that we see in one of the previous subsections when stacking more layers. Identity shortcut connections do not bring extra parameters nor computational complexity to the network.

In DnCNN the bypass is implemented for the hidden layers, so that 5 shortcuts with identity mapping used for 15 hidden layers, meaning that with every shortcut we pass three convolutional layers. The comparison between plain DnCNN and DnCNN with the bypass network structure placed in Table 3.1.

	DnCNN	DnCNN with bypass
Input	Grayscale image	
1st layer	Conv $\rightarrow$ ReLU	
bypass layer	-	Create bypass by copy the output of the previous layer
Block of 3 layers (2-3-4 layers)	Conv $\rightarrow$ BN $\rightarrow$ ReLU $\rightarrow$ Conv $\rightarrow$ BN $\rightarrow$ ReLU $\rightarrow$ Conv $\rightarrow$ BN $\rightarrow$ ReLU	Conv $\rightarrow$ BN $\rightarrow$ ReLU $\rightarrow$ Conv $\rightarrow$ BN $\rightarrow$ ReLU $\rightarrow$ Conv $\rightarrow$ BN $\rightarrow$ SumWith(Bypass) $\rightarrow$ ReLU
Repeat	Repeat previous two steps 4 times (for layers 5 - 16)	
17th layer	Conv	

Table 3.1. *Structure of DnCNN and DnCNN with bypass networks*

3.1.5 Inception layers

In basic DnCNN all convolutional layers comprised of small 3x3 kernels. What if the kernels are not all of the same sizes, but, keeping the total number of kernels per layer the same, one uses several kernel sizes, i.e. instead of 64 3x3 kernels in original DnCNN, use 16 kernels 1x1, 16 3x3, 16 5x5 and 16 7x7, or other combinations. The idea of using different kernel sizes is to extract features on a different scale in parallel. The approach of multiscale processing introduced by a team from Google [27]. The proposed method allows increasing the depth and width of the network by applying different size convolutional kernels. The concept of using kernels of different size in parallel illustrated in Figure 3.3 and also previously on the example of FLCNN in Figure 2.7.

When utilizing DnCNN with inception layers, the total number of layers is kept the same as in the basic DnCNN network ($N = 17$). Inception layers are utilized for the first layer and 15 hidden layers, while the last layer uses a 3x3 kernel to produce the output of DnCNN. The receptive field, in this case, can be calculated using the size of the largest kernel in the inception layer. Though, this is not fully appropriate, since kernels of different size used in parallel, and each kernel has a different receptive field.

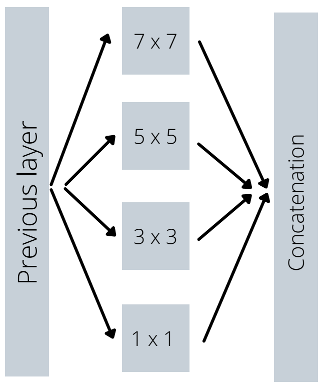

Figure 3.3. *Inception layer. Features from the previous layer are processed with convolutional kernels of different size.*

3.1.6 Inception layers comprised of small kernels only

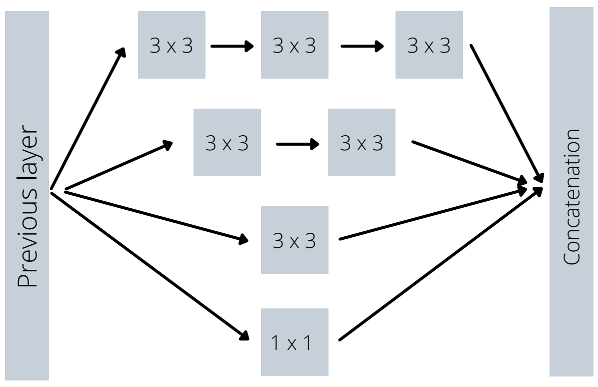

Figure 3.4. *Inception layers with smaller kernels. Features from the previous layer are processed with convolutional kernels of 3x3 size at maximum.*

The next step in studying the inception model is to utilize the receptive field property we discussed in subsection 3.1.1. What if instead of using big kernels, one replaces them with cascaded smaller kernels, i.e. 2 cascaded 3x3 kernels replacing a single 5x5 kernel?

This is similar to what is shown in Figure 3.1 where we demonstrated that kernel of any size can be represented in terms of the receptive field by stacking smaller kernels, which also lowers the number of learning parameters. In practice, the effective receptive field size of the cascade is smaller than that of the single kernel, but how much this affects the performance of the model?

In this experiment, the largest kernel size used is 3x3 kernels. So, 5x5 kernels were replaced with two consecutive 3x3 kernels, 7x7 kernels were replaced with three consecutive 3x3 kernels in order to significantly lower the number of computations while preserving the receptive field.

3.2 Experimental Numerical Results and Assessment

3.2.1 Training Environment

The networks in this work are implemented in Python using the Tensorflow framework [12] version 1.10 with Keras API [13] version 2.20. The models are trained on a server using AMD Ryzen Threadripper 1950X 16-Core Processor and GeForce GTX 1080 Ti graphics processing unit (GPU) on Ubuntu 18.04 LST operation system.

3.2.2 DnCNN setup

The adjustable parameters of the implemented DnCNN network when trained are the following: the number of epochs is 50, the batch size is 64 small grayscale images 64x64 pixels each, called patches, that are cropped randomly from the bigger color images of DIV2K training dataset [30] converted to grayscale. To increase the overall volume and diversity of the training data, each patch also can be augmented with a horizontal or vertical flip. The number of batches given to the network per epoch in the training process is 1024 (this is the number of steps per epoch). With these parameters, the accuracy of DnCNN is steady by the end of the training, so that the network with architecture as in Figure 2.5 saturates and does not learn any further, and therefore does not improve the accuracy at the end of the training. In each epoch, new images are presented to the network, which means that the overall number of images the network is trained on can be calculated as a product of the mini-batch size, steps per epoch, and epoch number. These three parameters can be adjusted so, that with more steps per epoch the mini-batch size should be changed correspondingly, given the epoch number fixed. The smaller mini-batch size will slightly affect the gradient descend step size so that to converge the network will need more steps.

In [32] two types of DnCNN networks are distinguished, DnCNN-B, trained on images, corrupted by the range of noise levels $\sigma \in [0, 55]$ (B is for Blind training, or altering noise level), and DnCNN-S network, trained on images corrupted by specific noise level (S in the name stays for Specific). DnCNN-S gives slightly better result over DnCNN-B when denoising images with $\sigma = 15$ and $\sigma = 25$, however, differences are negligible. In this work DnCNN will be referred to DnCNN-S, so that by default the networks have been trained and also tested on images corrupted with the specific noise level $\sigma = 25$, if not stated otherwise. The mean PSNR is **30.36** when tested on twelve grayscale images from dataset Set12 [23] shown in Figure 2.2.

3.2.3 Number of learnable parameters

In the following subsections, we will be experimenting with different network structures that will lead to a change in the number of the learnable parameters. It's important to understand how exactly the number of learnable parameters is calculated, as this is one of the characteristics we will be comparing across networks. We will now calculate it for DnCNN. The input layer contains a grayscale picture $N \times N \times C$, where C is the depth, equals to the number of channels ($C = 1$ for a grayscale image case), N is the size of the input (we consider the input to by a square image $N \times N$). The first convolutional layer extracts from the input 64 features, or feature maps. That requires 64 kernels of size $K \times K \times C + B$, where K is kernel size $K = 3$, C denotes the number of channels in the previous feature map (input image in this case) $C = 1$, B is a bias (constant) for each feature map, it brings one more weight for each feature map, so $B = 1$ in this sense. Substituting the numbers, we get $64 \times (3 \times 3 \times 1 + 1) = 640$ learnable parameters in the first convolutional layer. Layers 2-16 (fifteen hidden layers) extracts features from 64-channel data, so that the kernel size for them is $3 \times 3 \times 64$ plus one parameter for bias, giving for fifteen layers $15 \times 64 \times (3 \times 3 \times 64 + 1) = 553920$ trainable weights. In the end, the last convolutional layer extracts just one layer from the previous 64-channel feature map, having $1 \times (3 \times 3 \times 64 + 1) = 577$ parameters. In total, DnCNN with default settings operates with 555137 trainable weights. Now we understand how the number of learnable parameters is calculated. It depends on the kernel size, the number of layers, and the number of features, we extracted in each layer. More trainable parameters lead to a bigger capacity of the network but not necessarily to performance improvement. Additionally, the more weights a network needs to train, the more is the time needed to calculate them, and the more is the amount of GPU memory required.

In the previous paragraph, we counted the number of weights of the neurons as the number of learnable parameters (convolutional weights alone). However, worth to mention, that more parameters are participating in the overall process. There is also Batch Normalization (BN) operation and ReLU activation function applied after convolution operation within hidden layers in DnCNN as shown in Figure 2.5. BN is a normalization technique done between the layers, being applied to mini-batches (feature maps), not to the original input or to the last layer that produces the final output. Normalization is needed to avoid bias due to the differences in pixel values in the intermediate layers. It normalizes the range after convolution in hidden layers by subtracting the mean value and dividing by the standard deviation of the output of the neurons. In order to make the batch normalization work during training, the distributions of each normalized dimension need to be tracked. The BN parameters are also learned over the process (mean and standard deviation of the batch), aiming to decrease the loss of the model. Moving mean and moving variance are non-trainable parameters to propagate and backpropagate the information [17]. ReLU activation function outputs zero for all negative values or multiplies given values of normalized pixels in the feature map by a constant, that is a parameter of the activation function. In this work, we will concentrate our attention on the convolutional weights as the major contributing number of trainable parameters.

3.2.4 Receptive field

In the theoretical analysis subsection 3.1.1 we questioned how the performance of the network change when the depth is reduced and the kernel support is increased in such a way that the receptive field of the network remains constant? In this subsection we consider experimental setup for three kernel sizes $K = \{5, 7, 9\}$. For each kernel size, we derive the number of layers the networks need to have to satisfy the known fixed receptive field size ($RF = 35$). Note that in this experimental setup we use the corresponding kernel size only for hidden ($N - 2$) layers, while the kernels in the first and the last layers have fixed size $3 \times 3 \times C$, where C is the number of channels in the feature map on the previous layer. However, this is only the peculiarity of our experiment setup. In the end, we came to the following settings: for kernel size $K = 5$ we use $N = 9$ layers in total, for $K = 7$ we use $N = 7$, and for the biggest kernel $K = 9$ we construct the shallowest network with $N = 6$ convolutional layers. The results for the baseline DnCNN model ($N = 17, K = 3$) and its shallower versions are given in Table 3.2 below. The number of feature maps extracted in each layer (except the last one) is 64 for all setups. The number of parameters is calculated based on the experimental setup, considering the first and the last layers have fixed kernels $K = 3$.

Experiment	Layers N	Kernel size K	Features	RF	Parameters	sigma=25	sigma=50
100101	17	3	64	35	555137	30.36	27.15
100105	9	5	64	32	717952	30.30	27.02
100106	7	7	64	35	1004672	30.28	26.47
100107	6	9	64	37	1328256	30.27	26.07

Table 3.2. *Experiment results on the receptive field with DnCNN, mean PSNR for $\sigma = \{25, 50\}$ is provided.*

From the experiments, there is a clear PSNR decrease for shallower networks with bigger kernel sizes for both noise levels $\sigma = \{25, 50\}$. The best performance is achieved when DnCNN network configured with 17 layers using 3×3 kernel size. Using this network, mean PSNR value is 30.36 for twelve test images from Set12 (Figure 2.2), what is slightly lower that obtained in [32] (30.43 for noise-specific model DnCNN-S).

The network is trained on a batch of images, each batch comprised 64 patches. In this implementation patches of size 64×64 are adopted. The patches used for training are bigger than the receptive field area the network is looking at. Using patches smaller than the receptive field may not be useful. Patches 40×40 are used in original DnCNN-S and 50×50 in DnCNN-B [32].

In Figure 3.5 it is shown that the loss decays with more epochs. The decay is not steady at the beginning of the training (first 10 epochs), but then the loss decreasing steadily till the end of the learning process where it saturates and does not decrease much more. The loss decay is an important property of the learning process. It is done for every single experiment in this thesis so that all learning processes we are trying to establish are converging.

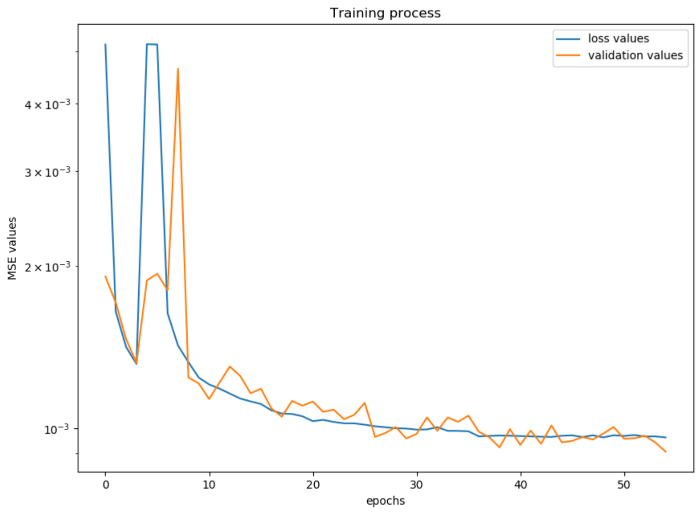

Figure 3.5. *Loss decrease with epochs for DnCNN with $N = 17$, $K = 3$ (100101 experiment in Table 3.2). $\sigma = 25$.*

3.2.5 Network depth

Seven networks were trained and tested in this experimental setup, where the layers varying from 7 to 34 as shown in Table 3.3. The training time increased for deeper networks as more parameters need to be optimally trained on the same amount of the training data. Here we explore the dependency between the mean PSNR value and the number of layers in DnCNN, given fixed kernel size (3×3).

From the experimental data in Table 3.3 we see that having more layers actually increasing the denoising ability of the models. For example, there is a solid improvement when using 17 layers compared to 6. However, after some point, this improvement is not significant. We even can see a performance drop for model using 34 layers $\sigma = 25$ and

Experiment	Layers N	Kernel size K	Features	RF	Parameters	sigma=25	sigma=50
100411	6	3	64	13	148864	29.88	26.42
100412	8	3	64	17	222720	30.12	26.70
100413	11	3	64	23	333504	30.28	26.99
100101	17	3	64	35	555137	30.36	27.15
100414	22	3	64	45	739712	30.37	27.08
100415	27	3	64	55	924352	30.39	26.93
100416	34	3	64	69	1182848	30.31	27.08

Table 3.3. *Experiment results on the number of layers in DnCNN with Set12, mean PSNR for $\sigma = \{25, 50\}$ is provided.*

for models with 22, 27, and 34 layers for noise level $\sigma = 50$. After some point, having more layers leads to decline in denoising performance for both noise levels $\sigma = \{25, 50\}$. In the experiment with 34 layers, the receptive field appeared to be bigger than the input patch size (64×64) we are training the network on. That means that the network tries to learn from the bigger input, while only the smaller patch is provided. That shows a very important role of the receptive field so that this parameter needs to be kept mind in all of the experiments.

The optimal number of layers for DnCNN is 17. With this number of layers DnCNN has moderate number of neurons, and receptive field is still within the size of the training input. This is also supported in [32], where a 17-layer network used for the DnCNN-S, trained with the specific noise level, as in our experiments ($\sigma = 25$), and 20 layers used for blind training in DnCNN-B with noise levels in range $\sigma \in [0, 55]$.

3.2.6 Feature extraction

Similar to increasing the number of layers, extracting more features requires more GPU memory to store and operate on weights and more computational time to upgrade weights. For the last experiment, where 320 features are extracted in each layer, the total number of parameters is so large, that the used GPU is not capable to hold it in memory. For that reason, in the last experiment, the mini-batch size was set to twice lower (from 64 to 32 images in one mini-batch), whereas the number of steps per epoch was increased two times (from 1024 steps in one epoch to 2048 steps) so that the overall amount of data given to the network during the training process stay the same. The training of the latter experiment took 20 hours.

According to the results in Table 3.4, higher PSNR obtained when more features were extracted. However, after some point, the PSNR value increases within the margin of error or even drops. The best result was obtained when extracting 64 features in each layer. With that, the balance between memory consumption, computational time, and the PSNR value was achieved.

Experiment	Layers N	Kernel size K	Features	RF	Parameters	sigma=25	sigma=50
100114	17	3	16	35	35105	29.79	26.48
100115	17	3	32	35	139329	30.09	26.88
100101	17	3	64	35	555137	30.36	27.15
100116	17	3	128	35	2216193	30.38	26.93
100117	17	3	320	35	13834881	30.36	27.31

Table 3.4. *Experiment results on the number of extracted features in DnCNN with Set12, mean PSNR for* $\sigma = \{25, 50\}$ *is provided.*

Within the last three subsections, we showed that the optimal values of the number of layers, the kernel size, and the number of extracted features were considered in the original DnCNN [32]. In three previous experiments we varied only one parameter, while fixed all of the others. This is similar to finding the partial derivatives of multivariable functions. Picking the best parameters leads to overall error minimization in our image denoising problem.

3.2.7 Identity shortcut connections

Experiment	N	K	F	RF	1.sigma=25	2.sigma=25	3.sigma=50	4.sigma=50
100118	17	3	16	35	29.79	29.90	26.48	26.53
100119	17	3	32	35	30.09	30.26	26.88	26.98
100101	17	3	64	35	30.36	30.39	27.15	27.12
100121	17	3	128	35	30.38	30.43	26.93	27.10
100123	17	3	320	35	30.36	30.36	27.31	27.18

Table 3.5. *Experiment results for DnCNN with and without use of shortcut connections, tested with Set12. Notations: N - number of layers, K - kernel size, F - number of features, RF - receptive field size, 1 - mean PSNR for DnCNN with noise* $\sigma = 25$*, 2 - mean PSNR for DnCNN with bypass with noise* $\sigma = 25$*, 3 - mean PSNR for DnCNN with noise* $\sigma = 50$*, 4 - mean PSNR for DnCNN with bypass with noise* $\sigma = 50$

When bypass added to the DnCNN structure and the network trained with the configuration from the previous subsection, we gain performance in all the experiments for $\sigma = 25$. In the case of a stronger noise level with $\sigma = 50$, the performance is increased for most of the setups, with the best result obtained when extracting 320 features. However, in this case, bypass brought a performance drop, while when extracting 128 features, its addition led to better denoising PSNR value. Results for DnCNN and DnCNN with bypass are shown in Table 3.5. Overall, using a shortcut connection yield a small improvement in all of the experiments.

3.2.8 Inception layers

Experiment	Layers N	Kernel size K	Features	Parameters	sigma=25	sigma=50
100201	17	1-3-5-7	16-16-16-16	1292160	30.44	27.29
100202	17	1-3-5-7	12-32-12-8	953888	30.42	27.25
100203	17	1-3-5-7	24-32-4-4	584864	30.39	27.22

Table 3.6. *Experiment results on the inception layers with DnCNN, mean PSNR for $\sigma = \{25, 50\}$ is provided.*

In this set of experiments, we use kernels of different sizes in parallel. In Table 3.6 the notation for kernel size 1-3-5-7 corresponding, for example, to the features 12-32-12-8, meaning that we use 12 kernels of size 1x1, 32 kernels of size 3x3, 12 kernels of size 5x5, and 8 kernels of size 7x7, all in parallel. We use different combinations of kernel sizes that all in total give 64 channels when we concatenate them. In this sense, we can say that the basic DnCNN has 64 3x3 kernels running in parallel. Basic DnCNN would correspond to kernels 1-3-5-7 and features 0-64-0-0 notation if it was in the results table. Having more smaller kernels leads to less learning parameters in the network.

DnCNN with incorporated inception layers provides better performance comparing to basic DnCNN for both noise models $\sigma = \{25, 50\}$ and any combination of the kernels. The best performance was achieved in the experiment when an equal number of all kernels were used (16 of each). The number of parameters used in the most robust inception version of DnCNN is significantly higher (more than twice) comparing to the plain DnCNN, which leads to a bigger size of the network and longer training time. Although, the inception model outperforms basic DnCNN even when it has a number of parameters comparable to base DnCNN in the case of 24-32-4-4 feature setup. Using inception layers leads to better performance for all tested proportions of the kernel size.

3.2.9 Inception layers comprised of small kernels only

Experiment	Layers N	Kernel size K	Features	Parameters	sigma=25
100301	17	1-3-5-7	16-16-16-16	438976	30.33
100302	17	1-3-5-7	12-32-12-8	466848	30.23
100303	17	1-3-5-7	24-32-4-4	372288	30.40

Table 3.7. *Experiment results on the inception layers inception layers with small consecutive kernels with DnCNN, mean PSNR for $\sigma = 25$ is provided.*

Table 3.7 summarizes the results of replacing 5x5 and 7x7 kernels with several consecutive 3x3 kernels in DnCNN with inception layers so that the receptive field is preserved. Here the notation for kernel size 1-3-5-7 corresponding, for example, to the features 12-32-12-8, meaning that we use 12 kernels of size 1x1, 32 kernels of size 3x3, 12 kernels of 2 consecutive 3x3, and 8 kernels of 3 consecutive 3x3, all in parallel, as shown in Figure 3.4. Using only smaller kernels drives to fewer training parameters. The performance of the model in this set of experiments is slightly worse comparing to the previous one. However, interestingly, the denoising numerical results here increase when a network contains fewer parameters and utilizing smaller kernels. The result for 24-32-4-4 features is similar to the best result in previous section, while the number of parameters here is more than three times less. This technique is used in FLCNN and showed excellent performance, where inception layers utilized after simple DnCNN-like layers.

3.2.10 FlashLightCNN depth

FLCNN is a network with a complex structure and its analysis was done in [5] and also studied in section 2.3.3. In this experiment only depth analysis will be performed with changes only in DnCNN-like convolutional layers.

Similarly to DnCNN network, we are going to vary the depth of FLCNN, keeping all the other parameters fixed. Here a notation for kernel size 3-5-[A]-[B] corresponding, for example, to the layers 5-4-3-2, meaning that we use 5 consecutive layers (Conv + BN + ReLU) with 3x3 kernels, then 4 consecutive layers (Conv + BN + ReLU) with kernels of size 5x5, next 3 layers implement inception module version A (Figure 2.7), and 2 layers implement inception module version B (Figure 2.8). In some of the experiments, we do not use layers with 5x5 kernels at all. The number of layers that contain inception modules will be fixed (always 3 and 2 for A and B respectively), while the DnCNN-like convolutional layers will be varied.

As an input, FLCNN is given a grayscale image of size 64x64, similarly to DnCNN. With an increase in the number of layers, the receptive field also increasing. In Table 3.8 it is shown that when the number of DnCNN-like layers with 3x3 kernels increased from 5 in basic configuration to 9, the receptive field, in this case, is bigger than the training image patch, and the performance dropped. The receptive field for the base FLCNN is just about the input image size. When increasing the number of layers with 5x5 kernels in the last three experiments, the performance stayed the same. Excluding layers with 5x5 kernels from the network, while having more layers with 3x3 kernels, provided a slight performance increase.

In our basic experimental setup (50 epochs with 1024 steps per epoch) mean PSNR for FLCNN is lower than in the paper [5]. This is due to the hyperparameters selection in the training process, as the experiments in this subsection were done with 1024 steps per epoch for 50 epochs, while the original FLCNN network has been trained with 2048 steps per epoch for 55 epochs. In other words, the original FLCNN was trained on more than twice more training data, which required an immense amount of time to train multiple experimental setups. We also conducted experiments training FLCNN for 55 epochs with 2048 steps per epoch. With these settings, the results stay high for any combination of

Experiment	Layers N	Kernel size K	RF	1.sigma=25	2.sigma=25
100500	5-4-3-2	3-5-[A]-[B]	65	30.38	30.52
100501	7-4-3-2	3-5-[A]-[B]	69	30.46	30.52
100502	9-4-3-2	3-5-[A]-[B]	73	30.22	30.49
100503	11-3-2	3-[A]-[B]	63	30.42	30.50
100504	13-3-2	3-[A]-[B]	67	30.41	30.49
100505	5-1-3-2	3-5-[A]-[B]	53	30.29	30.48
100506	5-2-3-2	3-5-[A]-[B]	57	30.29	30.52
100507	5-3-3-2	3-5-[A]-[B]	61	30.28	30.55

Table 3.8. *Experiment results on the number of different layer types in FLCNN, mean PSNR for $\sigma = 25$ is provided. 1 - mean PSNR when training for 50 epochs with 1024 steps per epoch, 2 - mean PSNR when training for 55 epochs with 2048 steps per epoch.*

the layers we used. Varying the number of layers, we were able to improve the denoising result comparing to the basic FLCNN configuration when trained on limited data (fewer epochs with fewer steps per epoch).

3.3 Conclusions to the chapter

In this chapter, we considered a simple network DnCNN to tackle the image denoising problem. Simple network structure allowed to study parameter contribution individually and independently. We presented a hyperparameter analysis of the network to seek performance improvements. For instance, the experiments showed that the receptive field size is a critical parameter that restricts the depth of the network. The network performance drops when the receptive field is larger than the input patch size used for training the network. The same-sized receptive field can be composed using smaller kernels applied consecutively in convolutional layers. This in turn leads to an increase in the network depth size. The benefit of using multiple layers in learning features at various levels of abstraction. Experiments showed the advantage and performance increase when using networks comprised of small kernels with such amount of layers so that the receptive field is about the training input size. We showed that with more layers, a network learns better up to a certain point, where the receptive field is getting larger than the input size. Adding more layers to the network does not boost the performance. Increasing the network depth also makes the number of learning parameters higher. The tradeoff between deep and wide networks needs to be found. Wide shallow networks are better at remembering, but not at generalization. Networks with more layers and smaller kernels are extracting information from the input on different scales, using intermediate features to generalize. Additionally, they have fewer trainable parameters when using small kernels versus when using large kernels. The is a high chance of overfitting for wide and deep networks. Also, the number of parameters for those will be so large, that it will take an immense amount of time and memory to train such a network. In real-life applications, the number of GPU memory and computational time is crucial when running models on e.g. a cellphone directly without using cloud resources. The numerous experiments on the hyperparameter analysis of the DnCNN show that its parameters are already optimally selected for the network architecture.

However, with the use of identity shortcut connection, we were able to improve the results: for DnCNN with fewer features, bypass provided gains in PSNR values from 0.1 to 0.2 dB; when more features are extracted in DnCNN layers, bypass still benefits the denoise task with smaller gains. Residual learning allows propagating the image itself, while the network operates on the noise rather than learning the image. The use of the inception layers provides multiscaling feature extraction and improves the denoising performance of the plain DnCNN.

4 USE OF TRANSFORMS WITH CNN FILTERING

Images in the spatial domain contain vast areas with similar pixel values. What if we first decorrelate the pixels before passing an image to the denoising algorithm? Providing less correlated features to the CNN denoiser may help a network distinguishing and learn the relevant peculiarities. In this chapter, we will alter the domain of the input images by applying different transformations. Therefore, in addition to CNN filtering, we analyze the role of the input data representation. First, we overview the theoretical basis for some transforms, in general, to understand how a sparse representation of the spatial image is done. In Chapter 2 we already familiarized ourselves with downsampling used in FFDNet and wavelet transform used both in WCNN and MWCNN. For wavelet transform analysis, we will consider two cases: the Discrete Wavelet Transform (DWT) and the Stationary Wavelet Transform (SWT), both using Haar wavelets. Here, we also cover single- and multi-level wavelet transform, DCT, and downsampling used in FFDNet. Moreover, we will introduce our variant of input data representation for image denoising, called muti-sigma noise estimation. Next, we conduct the experiments applying the transformations to the input of CNN (focusing mostly on DnCNN, but not limited to it). That means that the network structure stays unchanged, but the mapping the network learns is different. However, it is important to notice that we also considered various CNN architectures that work particularly with sparse representation. For example, WCNN is more similar to DnCNN architecture-wise with some enhancements like bypasses working with inputs in the wavelet domain. On the other hand, MWCNN architecture is distinctly specific to the wavelet transform. Transforms in these experiments are applied to the noisy input image and the ground truth image for CNN mapping. The goal is to study whether transforms affecting the denoising performance in terms of PSNR value or not. We will also study the CNN architectures more suitable for transformed data.

4.1 Discrete Cosine Transform (DCT)

Discrete Cosine Transform (DCT) is an invertible transform that is used widely in the digital image compression field. Joint Photographic Experts Group (JPEG) image compression standard uses DCT transform as a basis. From the frequency domain perspective, an image can be represented by the coefficients used in the decomposition of cosine functions of different frequencies. We will perform CNN learning process on the data in the frequency domain. Most of the energy of an image represented in the frequency domain concentrated in the few most significant coefficients in DCT, while the added noise pixels represent high frequencies we want to filter. The significant impact is established by lower frequencies in the frequency domain representation.

Technically, we can split an image into, for example, 2x2 non-overlapping blocks by using horizontal and vertical stride = 2 when the 2x2 window is sliding over the image. In the small 2x2 block all of the pixels are highly correlated. For each block, we compute DCT coefficients corresponding to basis vectors to decorrelate the features. The DCT coefficients in 1D are calculated using DCT II type definition in the Equation 4.1:

$$y_k = 2\sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi k(2n+1)}{2N}\right) \text{, multiplied by a scaling factor} \tag{4.1}$$

$$f = \begin{cases} \sqrt{\frac{1}{4N}}, \text{ if } k = 0 \\ \sqrt{\frac{1}{2N}}, \text{ otherwise} \end{cases}$$

In a 2D case, this is done by computing DCT for rows of the patch, transposing the result, and done DCT for the rows (former columns) again. This operation called 2D separable DCT.

The Inverse Discrete Cosine Transform (IDCT) is obtained in 1D using DCT III type definition:

$$y_k = \frac{x_0}{\sqrt{N}} + \sqrt{\frac{2}{N}}\sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi(2k+1)n}{2N}\right) \text{, multiplied by a scaling factor} \tag{4.2}$$

$$f = \frac{1}{\sqrt{2N}}$$

IDCT values are received in the same manner we got the DCT coefficients in a 2D case by consequent applying the Equation 4.2 to rows and columns of a produced estimate. For 2x2 pixel area DCT is the same as one-level Haar DWT that we will discuss in the following section devoted to the wavelet transform.

Worth to notice, that for each block we compute DCT coefficients using the same basis functions. One of the techniques we just want to mention here is Karhunen-Loeve Transform (KLT) [24] that provides the best decorrelation results because the transform itself considers the data values and builds a basis matrix for each block. This, of course, requires computing and storing the transform matrix for each block. DCT does not have such disadvantages and still approximates KLT well.

4.2 FFDNet downsampling transform

The pixel transformation used in FFDNet described very well in the corresponding subsection 2.3.4. A spatial image is reassembled to a 3D cube with 4 planes. The fifth plane added to the cube contains the noise level map. In our setup, the noise plane is uniform with all pixel values equal to the noise level σ. The downscaling procedure is shown in Figure 2.10. The approach of pixel organization vaguely resembles the arranging of RGB color filters in the Bayer filter mosaic. Mathematically, the pixel decomposition can be described as following:

$$F^0[c, x, y] = I\left[\left\lfloor \frac{c}{4} \right\rfloor, 2x + (c \text{ mod } 2), 2y + \left\lfloor \frac{c}{2} \right\rfloor\right],$$

where $0 \leqslant c < 4n_{ch}$, $0 \leqslant x < h$, $0 \leqslant y < \omega$, h - height of the spatial image, ω - width of the spatial image, c - channel number, n_{ch} - number of channels in image ($n_{ch} = 1$ in our grayscale case), F^0 denotes the result for zero layer (prior to non-linear transformations in CNN layers).

The upsampling for the four image estimate planes is depicted in Figure 4.1.

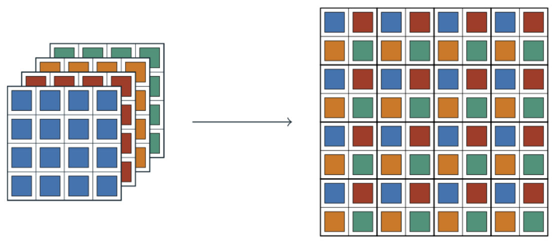

Figure 4.1. *Diagram of the FFDNet upsampling [28].*

4.3 Wavelet transform and Haar Wavelet

Wavelet transform already been mentioned in subsections 2.3.5 and 2.3.6. Here we would like to study how the wavelet transform is obtained. To approach the wavelet transform we first would like to step back to the major transform used in image processing since the late 1950s, the Fourier transform. The Fourier Transform using sine functions as the basis functions. The transform shows what frequencies a signal consists of. If we think for a second about the time signal (for example, sound), then the Fourier transform will only tell what are the frequencies (notes) present in the entire signal, but it will not tell us where they are located in the audio signal. To localize notes on the time axis, Short Time Fourier Transform (STFT) breaks the original time signal on small segments of equal time length, computing Fourier transform for each window separately. This allows us to see the distribution of used notes (frequencies) in time, not just their presence in the whole signal. One of the drawbacks of the STFT is that it has a fixed resolution.

Depending on the splitting window size, some low frequencies may not be detected correctly from the patch, being for example aliased. At the same time, high frequencies can vary in time a lot, having fast transients features (high-frequency content for short duration) within the same time window we use. The idea of using different window sizes for different frequencies lies in the wavelet method. Wavelet transform analyzes a signal at different frequencies with different resolutions. The higher the frequency the bigger the frequency range is used, as the Mel scale filter bank having a bigger range for high frequencies in one band. This is reasonable as having, for example, the uncertainty of 100 Hz may be acceptable for human auditory perception at a high frequency of 8000 Hz, but not around 400 Hz. The comparison between STFT and wavelet is exemplified in Figure 4.2. There it is shown that for wavelet frequency analysis the signal is divided into blocks, in each block, the higher is the frequency the bigger the frequency range and the shorter is the time window.

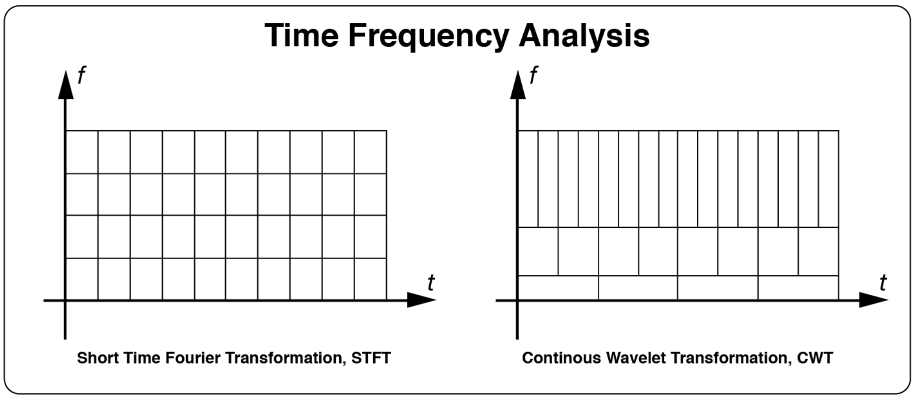
Time Frequency Analysis
f
t
f
t
Short Time Fourier Transformation, STFT
Continous Wavelet Transformation, CWT

Figure 4.2. *Time and frequency resolution in STFT and wavelet transform*

Images do not have a time dimension but contain segments with similar pixel information (low frequencies) and fragments with rapidly changing content, including edges (high frequencies). Wavelet transform is also applicable to show an image signal frequency-wise at different resolutions.

Haar wavelet is the first and the most comprehensible orthonormal wavelet. It is based on an orthogonal system of functions. Mother wavelet function $\psi(t)$ with zero value integral $\int_{-\infty}^{+\infty} \psi(t)dt = 0$ defines the signal details as follows:

$$\psi(t) = \begin{cases} 1, & 0 \leq t < \frac{1}{2}, \\ -1, & \frac{1}{2} \leq t < 1, \\ 0, & \text{otherwise.} \end{cases}$$

Where scaling function $\varphi(t)$ with unit integral $\int_{-\infty}^{+\infty} \varphi(t)dt = 1$ that determines the rough approximation of the signal, is constant:

$$\varphi(t) = \begin{cases} 1, & 0 \le t < 1, \\ 0, & \text{otherwise.} \end{cases}$$

Haar transforms for one-dimensional signal S is done like following. For each pair of adjacent elements we assign two numbers:

$$a_i = \frac{S_{2i} + S_{2i+1}}{2} \text{ and } b_i = \frac{S_{2i} - S_{2i+1}}{2}$$

Applying this to each element of the input signal, the result will contain two signals as the output. Each of them is twice less in length comparing to the original. One is a rough version of the input; the other bears the detailed information needed to reconstruct the original signal. The same Haar transform may be further applied to the obtained signals iteratively to form a mutiresolution picture of an image.

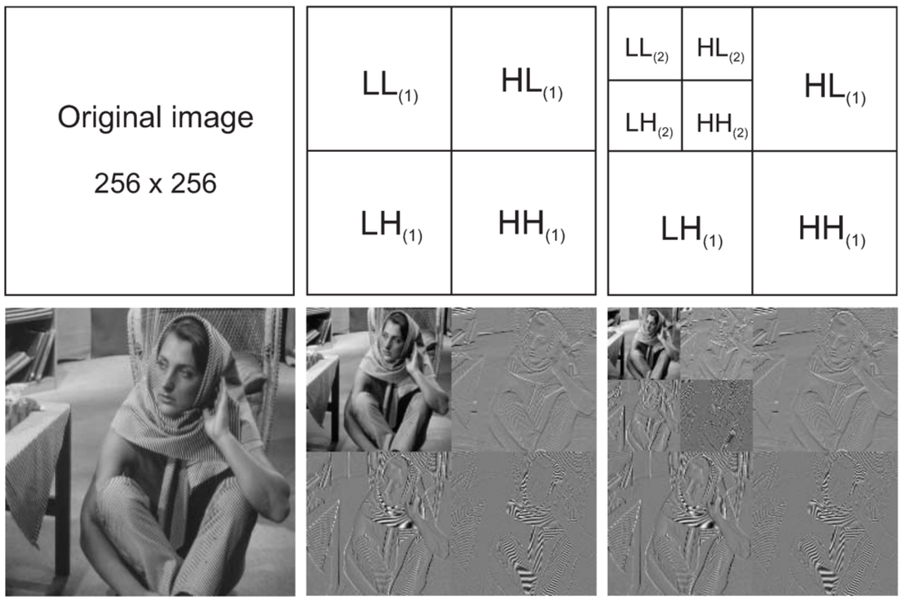

Figure 4.3. *Discrete Wavelet Transform (DWT) using the Haar wavelet.*

A two-dimensional Haar transform is a composition of one-dimensional Haar transforms. Let the two-dimensional input signal be represented by the matrix S. After applying the one-dimensional Haar transform to each row of the matrix S, two new matrices are obtained, the rows of which contain the approximated and detailing part of the rows of the original matrix. Similarly, a one-dimensional Haar transform is applied to each column of the obtained matrices, and four matrices are obtained at the output, one of which is an approximating component of the original signal, and the remaining three contain detailed information - horizontal, vertical, and diagonal. The result of applying the Haar wavelet transform to Barbara image from Set12 is illustrated in Figure 4.3. After the first iteration, the four image signals (LL, HL, LH, HH) are produced, each twice less in both dimensions of the original size. LL band contains the scaled downsampled version of the original image. The HL band contains the horizontal edges of the image, the LH block having the vertical edges. HH block contains high frequencies highlighting the diagonal edges. The second iteration is also shown in Figure 4.3 to illustrate the process can continue in general.

We also consider another wavelet transform with redundancies serve to overcome the lack of translation-invariance in DWT. This is achieved by upsampling the filter coefficients. Undecimated or Stationary Wavelet Transform (SWT) contains in the output the same number of pixels as its input. In the experimental section, we will examine how CNNs we discussed previously are dealing with input given in the wavelet domain, both in DWT and SWT using one-level Haar wavelet.

4.4 Multi-sigma noise estimation

The idea of multi-sigma noise estimation came from the fact that FFDNet is trained on images, corrupted with a noise model with variable standard deviation $\sigma \in [10, 75]$. With that, FFDNet is not a blind network and also requires knowing the noise level to eliminate that noise. Neither the basic DnCNN (DnCNN-S) that we are widely using is blind, meaning that it is trained on images corrupted by a specific noise level to deal with that noise level. And providing to such models images tainted by a different noise level will not improve the denoising quality, but only decrease it. Here in the thesis experiments, the vast majority of the networks are trained and tested using the noise level $\sigma = 25$. In contrast, FFDNet is trained with $\sigma \in [10, 75]$, and the blind version of DnCNN (DnCNN-B) is trained on $\sigma \in [0, 55]$. That allows to train these networks just once and then use them without re-training for images corrupted with the different noise levels, knowing the noise level (FFDNet), or even without that knowledge (DnCNN-B).

In the actual case applications, to achieve the best possible denoising result, we would need to know or somehow estimate the noise level of an image to use FFDNet, or DnCNN-S. In FFDNet case, we would provide a plane with the known noise level as the input; in DnCNN-S case, we would need to have a pre-trained model for that particular level of noise. We would like to construct another blind network based on the prosperous DnCNN that we will call multi-sigma noise estimation. What if we first train DnCNN-S separately for multiple noise levels, for instance, for seven sigma values, $\sigma =$ 10, 15, 20, 25, 30, 40, 80. Then we obtain models that deal with these noise levels the best. For a tainted input image with a generic noise level, we apply all pre-trained denoising models to form seven estimates, stack them together into a cube of estimates, which we call a multi-sigma cube of estimations. Depending on the input noise level, some of the models will remove noise better than the others, meaning that there should be one estimate that denoised better than all the others in terms of PSNR value. This is how we transformed a noisy input image into a cube of estimates for further processing. The next step is to construct a final estimate out of a multi-sigma cube image estimate representation.

Imagine for a second that instead of a cube of estimates obtained the way we described, we have a burst of images taken with a camera, capturing a static object. One of the simplest strategies for noise removal from real noisy observation (series of frames) is to average corresponding pixel values from the series of pictures. Another option would be to take a median pixel value within all seven pixels as a final pixel value. The randomly distributed AWGN is removed robustly this way. In our case, we can further employ CNN power to remove the noise from the multi-sigma cube of estimations.

Before providing the cube of estimates further to CNN, we stacked the initial noisy image itself into the cube. Instead of calculating the final pixel as a median value, we can train a network to find a combination of 8 images (7 estimates and the initial noisy image) for each pixel from the cube. This is done by employing non-linear layers with DnCNN using 1x1 2D convolution on the multi-sigma cube instead of 3x3 convolutional kernels used in the basic DnCNN. We train such DnCNN on the images contaminated with noise level $\sigma = 25$ for easy comparison to the previous results. The DnCNN model is trained for the residual case to learn the noise model and for the non-residual case. We will also consider a case with fewer estimates used in the multi-sigma cube, which should lead to a performance drop.

4.5 Experimental Numerical Results and Assessment

4.5.1 FFDNet downsampling transform

First, we use the basic DnCNN architecture with FFDNet downscaled input. In Table 4.1 we compare results obtained with different input setups and slightly different DnCNN architectures. We are using basic DnCNN with 17 convolutional layers overall that described in section 2.3.2. When using DnCNN with a bypass, it connects every third hidden layer, having five skip connections overall to propagate the identity through nonlinear hidden layers.

Experiment	Network	1. FFD 4-4 res	2. FFD 5-4 non-res	3. FFD 5-4 res	spatial
100620	DnCNN	30.03	29.82	30.06	30.36
100621	DnCNN with bypass	30.30	30.33	30.25	30.39

Table 4.1. *Experiment results for DnCNN with FFDNet input, tested with Set12. Mean PSNR for $\sigma = 25$ is provided. In case DnCNN with bypass, bypass passes every three hidden layers with 5 bypasses used in total. Notations: 1. FFD 4-4 res - FFDNet input (in this case only 4 bands, without noisy plane) inputted to DnCNN, in the last layer an estimate is subtracted with the noisy input (residual learning). 2. FFD 5-4 non-res - FFDNet input is given to DnCNN that outputs an image estimate (no subtraction layer, non-residual network). 3. FFD 5-4 res (noise) - FFDNet input provided to DnCNN that mapped to the noise we added, model learns noise model, not image model. Spatial - basic DnCNN with spatial input grayscale images.*

The most notable experimental result we see in Table 4.1, that incorporating the identity shortcut connections improves the denoising results in all cases. Bypass helps in both cases where a network model learns an image estimate (case 2) and when it learns a noise model (cases 1 and 3). Here in these experiments none of the setups with transformed input outperforms the basic DnCNN denoising performance. However, some of the results are very close to the basic DnCNN with spatial input.

Although the original FFDNet architecture is fairly similar to DnCNN, still there are differences. To see what settings are the most important, we decided to approach FFDNet setup gradually, by first trying FFDNet downsampling with DnCNN settings. The next step is to increase the number of epochs and set 80 epochs, as the original FFDNet using instead of 50 in DnCNN. For now, we will keep the mini-batch size the same as we used before in DnCNN (64 images in the mini-batch), while 128 is used in the original FFDNet. Steps-per-epoch increased three times to 3072 from 1024, however, the basic FFDNet uses as many as 8000 steps-per-epoch. Collectively the number of epochs and the number of steps-per-epochs increase the overall data provided to the network by $1.6 \times 3 = 4.8$ times, comparing to DnCNN. However, this is still significantly less than the basic FFDNet is using. In each epoch we now use $64 \times 3072 \approx 3 \times 64$K images, while in FFDNet in each epoch 128×8K $\approx 16 \times 64$K images are observed in the training. That means that in our next experiment we feed $16/3 \approx 5$ times less data, and our 80 epochs are equal to just $80/5 \approx 16$ epochs of the basic FFDNet in terms of the provided training data. The scheduling of the learning rate starts at 1e-3 for the first 50 epochs, then changes to 1e-4 for the following 10 epochs, and finally switches to 1e-6 for the remaining of the training. We performed multi-sigma training, adding different noise model σ from 10 to 75 to each patch, not just training on $\sigma = 25$ as in DnCNN case. Input is 5 channels with one noisy plane among them. FFDNet with 15 layers outputs an estimate, mapped to ground truth image (non-residual network). The result for this altered FFDNet setups provided in Table 4.2.

Experiment	Network configuration	sigma=25
100630	FFDNet lowered batch and steps	30.16
100631	FFDNet	30.35
100632	FFDNet with bypass	30.44
100634	FFDNet (17 layers) with bypass	30.48

Table 4.2. *Experiment results for FFDNet, tested with Set12. Mean PSNR for $\sigma = 25$ is provided. In case FFDNet with bypass, bypass passes every three layers with 5 bypasses used in total. Notations: FFDNet lowered batch and steps - 64 images in a mini-batch, 3072 steps-per-epoch. FFDNet (original) with settings from the original paper: 15 layers, 128 images in a mini-batch, 8000 steps-per-epoch, 80 epochs.*

The results in Table 4.2 show that the more training data we provide to the network, the better is the denoising performance of a model. Using lowered batch and steps setup provides even slightly worse result comparing to FFDNet input given to DnCNN in Table 4.1. Moreover, we were able to improve the FFDNet with original settings by adding identity shortcut connection into a network. Tuning the network further, we added more nonlinearity by injecting two extra convolutional layers.

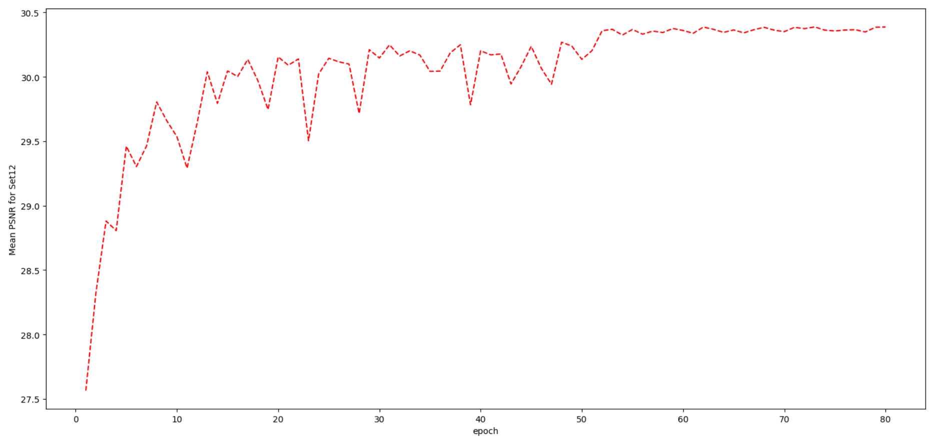

Figure 4.4. *PSNR stabilization with epochs for FFDNet with original settings. Each model tested on Set12 images, corrupted with sigma 25.*

In Figure 4.4 we show how the mean PSNR is stabilizing with epochs for FFDNet with original settings. This graphic shows that FFDNet stabilizes only after about 55 epochs, given the immense amount of training data provided in each epoch in comparison to DnCNN.

The transform that FFDNet is using allows achieving better results in image denoising. FFDNet needs much more training data to stabilize its denoising performance. Though once stabilized, it beats DnCNN with better denoising results. Also, the original FFDNet can be improved by adding shortcut connections and convolutional layers.

4.5.2 CNNs with DWT and SWT input

In the following experimental setups we will still use the basic DnCNN described in section 2.3.2 with a new input. As a network input, we provide a cube made from different bands of a wavelet transform. Initial spatial image is transformed into the wavelet domain, either DWT2 or SWT2 (2 after the transform name refers to the 2D transform). Decomposed images with 1-level Haar DWT2, resulting in 4 subbands - LL, LH, HL, and HH depicted in Figure 4.3. Stacking four subbands of the wavelet makes a cube that we provide as an input (similarly to WCNN shown in Figure 2.11). We train networks to produce four subbands in the output corresponding to the wavelet representation of a noise estimation model. To produce a recovered image estimate, the noise estimation wavelet cube is subtracted from the input, a wavelet cube representation of a tainted image observation. After applying IDWT2 to an image estimate in the wavelet domain, we get a denoised estimation of an image. As before, DnCNN with a bypass version connects every third hidden layer.

Experiment	CNN config	Input transform	Comment	sigma=25	sigma=50
100601	DnCNN	DWT2	64x64 input	30.23	26.92
100608	DnCNN	DWT2	128x128 input	30.17	-
100112	DnCNN + bypass	DWT2	64x64 input	30.28	27.17
100113	DnCNN + bypass	DWT2	128x128 input	30.28	-
100606	DnCNN	SWT2		30.31	27.13
100101	DnCNN	no transform		30.36	-
100610	DnCNN + bypass	SWT2		30.38	27.08
100613	DnCNN + bypass	SWT2	320 features	30.50	27.20
100602	FlashLightCNN	DWT2	64x64 input	30.18	-
100609	FlashLightCNN	DWT2	128x128 input	30.36	-

Table 4.3. *Experiment results on the different input transformations with DnCNN and FLCNN, mean PSNR for $\sigma = \{25, 50\}$ is provided.*

From the experiment results listed in Table 4.3, we see that using wavelet transform input to DnCNN improves the denoising capacity over the spatial case. When training in the spatial input case, we used 64x64 pixel images. When transforming such a patch into a DWT cube input, its size is halved (32x32x4). SWT preserves the input size and transforms 64x64 input into a 64x64x4 cube. The receptive field of DnCNN is 35x35 pixels, which is a bit bigger than the input size in the DWT case. Remembering the importance of the receptive field, we also conducted experiments with twice larger input patches, 128x128 pixels, so that after the DWT the size will still be larger than the receptive field size. Results show that doubling the input patch size does not make any difference for DnCNN with DWT input. Both DnCNN with and without shortcut connection have similar denoising results for regular and enlarged patches. However, adding a bypass slightly improves the performance. Using SWT with redundant image data beats the results where DWT is used. DnCNN with bypass and SWT input outperform the basic DnCNN. One of the experiments illustrates the performance with an intense number of features extracted with non-linear layers, 320 features instead of 64 (number of features inspired by WCNN). Individually, increasing the number of features does not give any improvements, however when used with bypass and SWT input, it gains 0.12 dB over the same settings with fewer features extracted. The combination of using DnCNN with bypass, SWT input, and extreme amount of features extracted, provides outstanding performance in this case with PSNR of 30.50.

We also conducted experiments on FLCNN with DWT input. Since the receptive field of FLCNN is 65x65 pixel area, even larger than for DnCNN, the patch size increase is essential. Experiment shows that doubling the patch size from 64x64 to 128x128 pixels benefits the performance significantly in this case. This again proves the high importance of the receptive field size of a network.

4.5.3 Multi-sigma noise estimation

The first preparation towards the implementation of the multi-sigma noise estimation idea is to train DnCNN with multiple noise levels. We trained separately with seven sigmas $\sigma = 10, 15, 20, 25, 30, 40, 80$, with fewer steps for lower noise, and bigger steps for high noise. The number of pre-trained cases can vary in general. In Table 4.4 PSNR column provides the results for DnCNN that was trained and tested on the same sigma value that specified in the corresponding row. DnCNN learns well how to denoise images corrupted by minor noise made with $sigma = 10$. For stronger noise, the neural network learns less well. Testing all seven models against Set12 images tainted with $\sigma = 25$ shows that the best denoising result is obtained when the model was trained with the same noise used when testing. That illustrates each of the seven models works best with the images corrupted by the noise they were trained on.

CNN	sigma	PSNR	sigma=25
DnCNN	10	34.74	21.68
DnCNN	15	32.78	23.54
DnCNN	20	31.45	27.48
DnCNN	25	30.31	30.29
DnCNN	30	29.49	29.70
DnCNN	40	28.19	28.20
DnCNN	80	24.96	24.74

Table 4.4. *Experiment results for DnCNN trained on images corrupted with specific sigma noise, $\sigma = \{10, 15, 20, 25, 30, 40, 80\}$. PSNR column provides the results of testing Set12 images tainted with the corresponding noise level, same as used when training. Sigma=25 column provides the results of testing Set12 corrupted with $\sigma = 25$ against the models trained with corresponding noise level.*

Experiment	CNN	cube of sigmas	kernel	sigma=25
100901	no network	10, 15, 20, 25, 30, 40, 80	-	35.51
100901-2	no network	10, 15, 20, 25, 30, 40, 80	-	35.30
100902	DnCNN	10, 15, 20, 25, 30, 40, 80	1x1	30.36
100903	DnCNN	10, 15, 20, 25, 30, 40, 80	1x1	30.34
100904	DnCNN	20, 25, 30	3x3	29.36

Table 4.5. *Experiment results for DnCNN trained on a multi-sigma cube of estimates. PSNR values provided when tested on Set12 images corrupted with noise level $\sigma = 25$.*

Before setting training on multi-sigma cube of estimations, we conducted experiments that do not use CNN for a sanity check to see the concept makes sense. In Table 4.5 first two rows provide the results of experiments where the final estimate is constructed by picking the pixel value from the multi-sigma cube of estimates that is closest by value to the clean ground truth image so that the final estimate is constructed only of pixels from the cube without any non-linear modifications. In the first experiment the noisy image itself was also stacked into the cube, so the cube contains 8 planes, while in the second experiment only seven estimates are used. The results in these two cases show that the final image can be reconstructed with good precision in terms of PSNR value, so we now try to train DnCNN to do the job. The following two networks were trained with a cube of 8 planes, 7 estimates, and the noisy image itself. We trained DnCNN that use 1x1 convolutional kernel size in all layers. Two networks were trained, a network that learns the image model, resulting in PSNR 30.36, and a network that learns the noise model with PSNR 30.34. These results are comparable with results of a basic DnCNN-S and also to DnCNN-B network, meaning that this approach to the input data works fine. Also, we experimented with just three estimates for $\sigma = 20, 25, 30$ to test with sigma=25, using the basic residual DnCNN that learns noise model with 3x3 convolutional kernels. In this case, the result of testing Set12 tainted with noise level sigma=25 provided inferior results comparing to a higher number of estimates in the input.

4.6 Conclusion to the chapter

In this chapter, we discussed and examined how an input image representation affects the CNN denoising capabilities.

The experimental results on the input transformation show that some sparsifications of the input may help the learning process and leads to better denoising. For example, the performance of the DnCNN improved over 0.1 dB when SWT was applied to the input patch, considering the advantage of bypass and extensive feature extraction. Using different sparsifications for FLCNN does not provide practical enhancements, i.e. it seems that this network only works well with spatial-domain input images. Downsampling transformation used in FFDNet does not achieve better PSNR with the baseline DnCNN settings. Nevertheless, with more epochs and steps per epoch, FFDNet with bypass achieved the significant denoising result, beating DnCNN by 0.1 dB. The proposed multi-sigma noise estimation achieved similar performance to DnCNN, showing it is promising.

We consider it important to describe also the cases that do not lead to better denoising performance. The ideas behind those are still interesting to discuss and the results can help with future experimentations.

The sparsification can help in image denoising. The experiments showed that the use of transformation with CNN filtering improves the overall denoising performance. Using different input image representation may require different CNN configuration. Also, bypass connections proved to be practical in the experiments. When changing the input domain, it is also crucial to remember the receptive field concept.

5 USE OF TRANSFORMS AND NON-LOCAL FILTER WITH CNN FILTERING

In the previous chapter, we showed that pixel decorrelation prior to the CNN filtering is effective for noise removal problems. In some cases, providing a transformed image improved the denoising capacity of a network significantly. In most other cases, the result was not worse than when providing a spatial image.

A paradigm of independent iterative use of CNNF and NLF was covered in section 2.4 under the NN3D name. In this chapter, we investigate if the performance of the combination of transformation (sparsification) of the input images and local denoising by CNN can be further enhanced by NLFs.

The strategy of NN3D filtering does not require training a new network, it is a plug-in approach where a required model is plugged. We will use the pre-trained CNN models. Networks that were exploited above will be now re-evaluated using NN3D architecture.

5.1 Experimental Numerical Results and Assessment

For simplicity, in our experiments with NN3D, we will use just two iterations. On the first iteration, the polluted image with noise level σ is given to NN3D. On the second iteration, we consider that the new input image is a mix of the initial noisy image and an estimated recovered image obtained after the first iteration. We choose parameters so that the noise level of this estimated clean image is supposed to be $\sigma/2$. For each experiment in this section, we would need to have two pretrained CNNs, one for the sigma value σ, and another one for the $\sigma/2$ noise level used on the second iteration of NN3D.

First, we will try NN3D on DnCNN and FLCNN with spatial input images. Then, we exploit the whole stack, using CNNF with sparsified input and power of NLF, for instance, applying NN3D for DnCNN with input in DWT. FFDNet uses its own sparsification for the input image that we do not specify that separately.

The results depicted below in Table 5.1 show that the performance of DnCNN and FLCNN networks can be further improved by using NLF, incorporating these CNN models as a part of NN3D architecture. For DnCNN with spatial input the results for using iteratively with NLF show significant improvement for more than 0.1 dB, working especially robust for higher noise $\sigma = 50$. When denoising images corrupted with noise level $\sigma = 25$ on the second iteration model trained for $\sigma = 10$ is used instead of the required $\sigma = 25/2 = 12.5$ as we pick a model that we have trained already that is close enough. The exception is FFDNet that is trained once for the range of noise $\sigma \in [10, 75]$, so we are having models trained for all desired sigmas in that range, including $\sigma = 12.5$. FLCNN plugged into NN3D also was improved for both its full and light versions. DnCNN with input in DWT improved on about 0.15 dB when integrated to NN3D for high noise level $\sigma = 50$. NN3D with FFDNet CNN block performs as well as the FFDNet itself. The only case where FFDNet with NN3D underperforms the FFDNet itself is when denoising applied for input image with noise level $\sigma = 10$, so that on the second iteration $\sigma = 5$ is used, but the FFDNet trained only on the range of $\sigma \in [10, 75]$.

CNN config	sigmas used	CNN PSNR	NN3D(CNN) PSNR
DnCNN, no input transform	25, 10	30.36	30.45
DnCNN, no input transform	50, 25	27.15	27.29
DnCNN, DWT input with bypass	50, 25	27.16	27.29
FLCNN light, no input transform	25, 10	30.38	30.48
FLCNN, no input transform	25, 10	30.52	30.57
FFDNet [17 layers with bypass]	10, 5	34.63	34.49
FFDNet [17 layers with bypass]	25, 12.5	30.50	30.50
FFDNet [17 layers with bypass]	30, 15	29.69	29.70
FFDNet [17 layers with bypass]	50, 25	27.42	27.46

Table 5.1. *Experiment results on the different CNNs and NN3D approach with the same CNN plugged in. Mean PSNR for the highest sigma value from the list is provided when tested on Set12. In case DnCNN and FFDNet with bypass, bypass passes every three hidden layers with 5 bypasses used in total. FLCNN light is using 1024 steps per epochs and 50 epochs, while FLCNN using 2048 steps per epoch and 55 epochs.*

In general, some other proportion of images (parameter λ in NN3D) can be set to seek better results. Also, more iterations in NN3D can be explored.

5.2 Conclusion to the chapter

In this chapter, we investigated how CNN can be further improved with NLF applied afterward, using as input both spatial and sparsified images. Iteratively applying non-linear CNNF and NLF, the denoising result improved in all setups. Especially noticeable improvement in PSNR value we spot for the DnCNN and FLCNN. The denoising capacity of FFDNet stays at the same high level when using NLF scheme after the CNNF block. Input image representation affects the NN3D denoising capabilities compared to the CNN with sparce image alone. With that, PSNR value for DnCNN with DWT input setup improved on about 0.15 dB with NLF part and two iterations in NN3D.

Using all three components, input image transform, CNNF, and NLF by using NN3D, improves the results of NN3D alone without image transformation.

6 CONCLUSION

The thesis work provides rich experimentally based research and analysis in the image denoising field. In this work, we have considered a number of different methods and examined actual denoising algorithms from the state-of-the-art articles. As a result of the parameter analysis, the vast majority of the learning-based algorithms and deep networks were improved in terms of denoising capability. In the thesis, essential parameters of the neural networks are studied thoroughly to reveal their individual impact while seeking better denoising results based on PSNR value.

One of the possible continuations of the study is to consider and analyze the denoising quality when some classification of the input image is done prior to denoising. With that strategy, images that are classified as having strong self-similarity would be processed with CNNF and NLFs, as conducted in Chapter 5, while images that are classified as having low self-similarity would be processed with local denoising by CNNF only.

REFERENCES

[1] M. Aharon, M. Elad and A. Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *Signal Processing, IEEE Transactions on* 54 (Dec. 2006), 4311–4322. DOI: 10.1109/TSP.2006.881199.

[2] B. Ahn and N. I. Cho. Block-Matching Convolutional Neural Network for Image Denoising. (Apr. 3, 2017). arXiv: http://arxiv.org/abs/1704.00524v1 [cs.CV].

[3] Arnault. *About Convolutional Layer and Convolution Kernel*. 2018. URL: https://www.sicara.ai/blog/2019-10-31-convolutional-layer-convolution-kernel.

[4] W. Bae, J. Yoo and J. C. Ye. Beyond Deep Residual Learning for Image Restoration: Persistent Homology-Guided Manifold Simplification. *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (July 2017). DOI: 10.1109/cvprw.2017.152. URL: http://dx.doi.org/10.1109/CVPRW.2017.152.

[5] P. H. T. Binh. Deep Neural Networks for Image Denoising. MA thesis. Tampere, Finland: Tampere University, 2020.

[6] A. Buades, B. Coll and J.-M. Morel. A non-local algorithm for image denoising. *IN CVPR*. 2005, 60–65.

[7] C. Cruz, A. Foi, V. Katkovnik and K. Egiazarian. Nonlocality-Reinforced Convolutional Neural Networks for Image Denoising. *IEEE Signal Processing Letters* 25.8 (Aug. 2018), 1216–1220. DOI: 10.1109/lsp.2018.2850222.

[8] K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian. Image denoising with block-matching and 3D filtering. *Image Processing: Algorithms and Systems, Neural Networks, and Machine Learning*. Vol. 6064. International Society for Optics and Photonics. 2006, 606414.

[9] K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian. Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 16 (Sept. 2007), 2080–95. DOI: 10.1109/TIP.2007.901238.

[10] K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian. A nonlocal and shape-adaptive transform-domain collaborative filtering. *Proc. Int. Workshop on Local and Non-Local Approx. in Image Process., LNLA*. 2008.

[11] K. Dabov, A. Foi, V. Katkovnik and K. Egiazarian. BM3D image denoising with shape-adaptive principal component analysis. *SPARS'09-Signal Processing with Adaptive Sparse Structured Representations*. 2009.

[12] François Chollet. *Tensorflow*. 2015. URL: https://www.tensorflow.org.

[13] Google Brain Team. *Keras API*. 2015. URL: https://keras.io.

[14] S. Gu and R. Timofte. A Brief Review of Image Denoising Algorithms and Beyond. Jan. 2019, 1–21. ISBN: 978-3-030-25613-5. DOI: 10.1007/978-3-030-25614-2_1.

[15] S. Gu, L. Zhang, W. Zuo and X. Feng. Weighted Nuclear Norm Minimization with Application to Image Denoising. *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2014. DOI: `10.1109/cvpr.2014.366`.

[16] K. He, X. Zhang, S. Ren and J. Sun. Deep Residual Learning for Image Recognition. (Dec. 10, 2015). arXiv: `http://arxiv.org/abs/1512.03385v1 [cs.CV]`.

[17] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: `1502.03167 [cs.LG]`.

[18] J. Jordan. *Common architectures in convolutional neural networks*. 2018. URL: `https://www.jeremyjordan.me/convnet-architectures/`.

[19] A. Krizhevsky, I. Sutskever and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems* 25 (Jan. 2012). DOI: `10.1145/3065386`.

[20] B. Liu and J. Liu. Overview of Image Denoising Based on Deep Learning. *Journal of Physics: Conference Series* 1176 (Mar. 2019), 022010. DOI: `10.1088/1742-6596/1176/2/022010`. URL: `https://doi.org/10.1088%2F1742-6596%2F1176%2F2%2F022010`.

[21] P. Liu, H. Zhang, K. Zhang, L. Lin and W. Zuo. Multi-level Wavelet-CNN for Image Restoration. (May 18, 2018). arXiv: `http://arxiv.org/abs/1805.07071v2 [cs.CV]`.

[22] W. Luo, Y. Li, R. Urtasun and R. Zemel. *Understanding the Effective Receptive Field in Deep Convolutional Neural Networks*. 2017. arXiv: `1701.04128 [cs.CV]`.

[23] [online]. *Set12, BSD68 and other datasets*. Available: https://github.com/cszn/FFDNet/tree/mas 2017.

[24] E. K. R. Rao and P. C. Yip. The Transform and Data Compression Handbook. *in 1994, and the M.S.E. degree from the University of Pennsylvania, Philadelphia, in 1996. Both in electrical engineering. He received his Ph.D. in systems engineering from the University of Pennsylvania in*. CRC press, 2001.

[25] O. Ronneberger, P. Fischer and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. (May 18, 2015). arXiv: `http://arxiv.org/abs/1505.04597v1 [cs.CV]`.

[26] I. Shafkat. *Intuitively Understanding Convolutions for Deep Learning*. 2018. URL: `https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1`.

[27] C. Szegedy, S. Ioffe, V. Vanhoucke and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. (Feb. 23, 2016). arXiv: `http://arxiv.org/abs/1602.07261v2 [cs.CV]`.

[28] M. Tassano, J. Delon and T. Veit. An Analysis and Implementation of the FFDNet Image Denoising Method. *Image Processing On Line* 9 (2019), 1–25. DOI: `10.5201/ipol.2019.231`.

[29] P. H. Thanh Binh, C. Cruz and K. Egiazarian. Flashlight CNN Image Denoising. *arXiv e-prints*, arXiv:2003.00762 (Mar. 2020), arXiv:2003.00762. arXiv: `2003.00762 [eess.IV]`.

[30] R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang, B. Lim et al. NTIRE 2017 Challenge on Single Image Super-Resolution: Methods and Results. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. July 2017.

[31] Z. Wang, A. Bovik, H. Sheikh and E. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13.4 (2004), 600–612. DOI: `10.1109/TIP.2003.819861`.

[32] K. Zhang, W. Zuo, Y. Chen, D. Meng and L. Zhang. Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising. *IEEE Transactions on Image Processing* 26.7 (July 2017), 3142–3155. DOI: `10.1109/tip.2017.2662206`.

[33] K. Zhang, W. Zuo and L. Zhang. FFDNet: Toward a Fast and Flexible Solution for CNN based Image Denoising. (Oct. 11, 2017). DOI: `10.1109/TIP.2018.2839891`. arXiv: `http://arxiv.org/abs/1710.04026v2 [cs.CV]`.