Fauxtoshop: Modeling Image Editing Operations with Kernel Prediction Networks and Parameter Blocks

Ziyin Ma

May 2020

Contents

Abstract

This work explores various way of building a proxy function for image editing software. We introduce parameter blocks that can be used with kernel prediction network as a highly effective proxy model. We show it is critical to disentangle predicting the operations and applying them on images. Specifically, we show with various experiments that our model is 1) more effective and more robust than other models 2) more interpretable with explicitly predicted operations to be applied on images 3) using less parameters than other models. This work also serves as an important early exploration in building a framework for transferring between image editing software while preserving non-destructive edits through parameter mapping.

1 Introduction

Image editing software provide users with a large collection of editing parameters like exposure, shadow, etc. These parameters are often represented by a scalar value or a slider on the user interface. The combination of editing parameters allows users to produce desired images with simple control.

We observed that (Section 7) there is a lack of consistency on the implementation of editing parameters across software. The name and number of editing parameters found in software vary greatly. Moreover, modern software applies **content-aware and spatially-varying** editing on raw images for artistic editing effect. Therefore, it is almost impossible to provide a direct mapping between the parameters used between a pair of software.

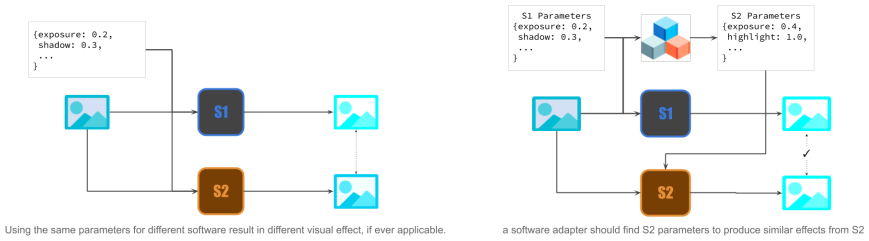
These complications make it very difficult for users to migrate from one editing software to another while preserving all the edits. Ideally, a software adapter $\mathcal{F}$ should create a parameter mapping between the source P_{source} and the target software P_{target}. Given that the effect of the parameter are content-aware, the software adapter would also need to be aware of the original image I.

$$\mathcal{F} : (P_{source}, I) \rightarrow P_{target}$$

P_{target} should achieve similar visual effect in the target software that P_{source} achieves in the original software. (Figure 1)

There are many ways to build a deep learning model that serves as $\mathcal{F}$. A possible approach is shown in Figure 2 where a proxy function is trained to simulate the behavior of the target software. Given such a proxy function, a second model can be trained to map parameters in the source software to the target software.

At the heart of many lies a *differentiable proxy function* $\mathcal{P}$ that can replace the source or the target software. Most image editing software are not written in differentiable language or not open-sourced at all. We explored many aspects of creating a proxy function for image editing software RawTherapee. This work proposes a variant of kernel prediction network [1] that achieves satisfactory performance as a proxy function for RawTherapee. When compared with a direct use of U-Net:



(a) (b)

Figure 1: (1a) Using the same parameters for different software result in different visual effect, if ever applicable. (1b) A software adapter should find S2 parameters to produce similar effects from S2

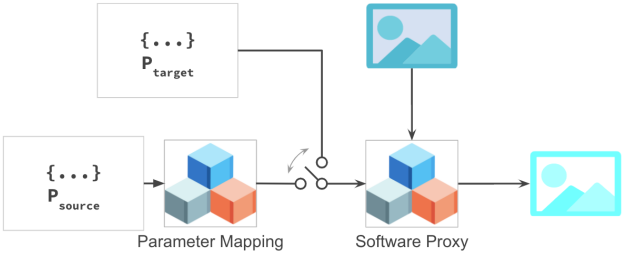


Figure 2: A possible deep learning model for trainable software parameter mapper

- Our kernel prediction network consistently outperforms U-Net on all sorts of real and artificial tasks.
- The parameter blocks we proposed improved model performance by a significant margin.
- Parameter blocks also reduce the number of parameters in the model.
- We illustrate through experiments that the two-step approach taken by the kernel prediction network poses a healthy and helpful regularization on the model.

2 Related Work

2.1 Exposure: A White-Box Photo Post-Processing Framework

[2] tackles automatic post-processing by predicting a sequence of parameterized editing operations like exposure, gamma, color curve, etc. It applies CNN networks on a low-resolution (64×64) version of the input image. The policy network proposes one of the predefined operations and predicts its associated parameters. The output is put back into the policy network to predict the next operation. The model is considered a "white-box" in that it can provide a interpretable sequence of operations for automatic post-processing. The very same architecture is used for value network and critic network to create reinforcement feedback.

The problem we are trying to solve is different from theirs, and many auto-editing methods, where we can take arbitrary parameter inputs and simulate software behavior while the literature mentioned above focuses on choosing one set of parameters for the best artistic effect. Nevertheless, this work shed light on the fact that image editing often involves global operations that U-Net-like architecture won't perfectly capture and might require further design. This work also proposes the general pattern of handling parameters in CNN network-appending parameters as extra channels.

2.2 Why Kernel Prediction?

[1] proposes a burst denoising network for predicting a set of convolution kernels that merges a stack of burst images. A unique convolution kernel is predicted for each pixel in the eventual output. Generating images in an end-to-end fashion is simple, yet less interpretable and less regularized. Predicting operations (the kernels) based on input image and input parameters fits our project better both empirically and theoretically.

To prove this, consider a trivial software that can only apply affine transformation on input image I. Output Y can be described as $Y = k \times I + b$ where k and b are the input to the trivial software. We now compare using an end-to-end U-Net and using a U-Net to predict the kernels to be applied on I. To make the network aware of k and b, we expand and concatenate them as extra channels to I so that the input becomes $I' = (I; k; b)$.

Since a kernel prediction network needs to predict affine kernels to be applied on I, that kernel can simply be k and b. This only requires one layer at minimum, namely, the network should select the extra input channel as its predictions. For the end-to-end U-Net, the task becomes harder. It will have a difficult time multiplying the values from two channels of its input, as both I and k are in the input. It can be concluded this cannot be achieved in one layer for end-to-end U-Net.

The paper takes burst images and produce one final images with high quality. The kernel is applied on the channel across all input images. Our approach is

more general in that we allow each channel in the output image to be generated with a convolution over all channels in the input image.

[3] is a follow-up to [1] and predicts multiple kernels of various sizes for each pixel in input images.

2.3 Replacing Mobile Camera ISP with a Single Deep Learning Model

[4] proposes a novel pyramidal CNN architecture that can replaces camera ISP. Images are down-sampled to multiple scales and are passed through parallel residual blocks. The outputs are then gradually up-sampled and merged into the final output. The model adopts different losses on different scales and is trained from low levels to higher ones.

3 Method

3.1 Proxy Functions for Editing Software

We experimented with three approaches to building a proxy function $\mathcal{P}$ that maps the input image and parameters to an output images that resembles software output.

An end-to-end U-Net. U-Net is widely used for image generation tasks that originates from another image. It captures both local and global features with its intermediate layers of varying scale. It also avoids loss of information of intermediate layers by maintaining copying links. This fits our assumption that image editing are both content-aware locally and globally. To make the network aware of the parameter input, we append the input image I with extra channels filled with input parameters p.

Theoritically, U-Net alone would be sufficient for being a proxy function for image editing software. However, we will show that this architecture has its limits in finding the correcet operation based on the image and applying the operations in an end-to-end fashion. More precisely, we believe that these two steps can be disentangled which will benefit image generation. This motivates the following two ideas.

An one-time kernel prediction network. A kernel prediction network [1] utilizes U-Net in a similar fashion. The output of the U-Net is not considered the final output. Rather, it is used as the per-pixel kernel and bias to be applied on the input image. The final output of the architecture is the output of applying this kernel and bias on the input image. (See Figure 3)

A kernel prediction network disentangles predicting the correct operations to use and applying such operations on the input image. The kernel and bias being predicted is the operation to be applied on the input image later. Experiments show that this is a strong regularization that result in higher performance.

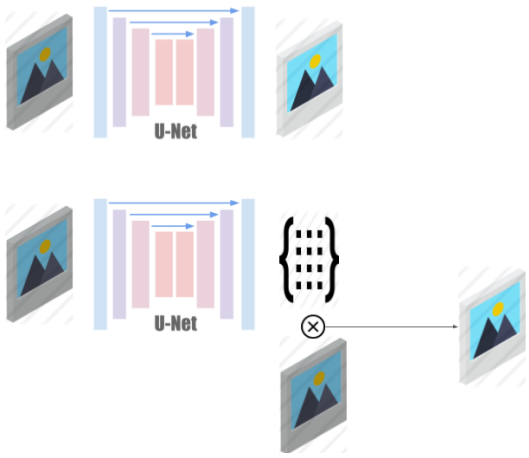


Figure 3: End-to-end prediction vs. predicting operations to be applied

A hybrid kernel prediction network. The kernel prediction network from [1] only predict kernels once. This limits the operation predicted to be an one-time linear operation. To enable the model to apply arbitrary number of operations, we introduce a hybrid kernel prediction layer as a building block of a hybrid kernel prediction network.

It uses a fully connected layer to predict a kernel to be applied in convolution layers. (See Figure 4.) The input to the connected layer can be the input parameters or that combined with a neighbourhood in input response map. The major difference between a hybric kernel prediction layer and a common convolution layer is that a static kernel is stored in a common convolution layer. However, the kernel being used is dynamically predicted in a hybrid kernel prediction layer.

We can freely combine kernel prediction layers and common convolution layers. The user only need to specify the desired number of channels in the output.

3.2 Incorporate Parameters Input into CNN Architecture

In the default setting, we append parameters as extra channels to the input image and activation maps at each layer. This is an intuitive approach. However, if kernels of non-trivial size are used, the same value would appear repeatedly in their receptive fields. For example, a 3×3 kernel will see the same value in the parameter channels 9 times. This introduces unnecessary linear correlation and redundant parameters in the trained kernels.

To reduce such redundancy, we propose parameter blocks. The block consist of two convolution layers (Figure 5). The first is a regular 3×3 convolution where no parameter is appended. It will transform local features into scalar values while preserving the number of channels. The second layer is a 1×1 convolution

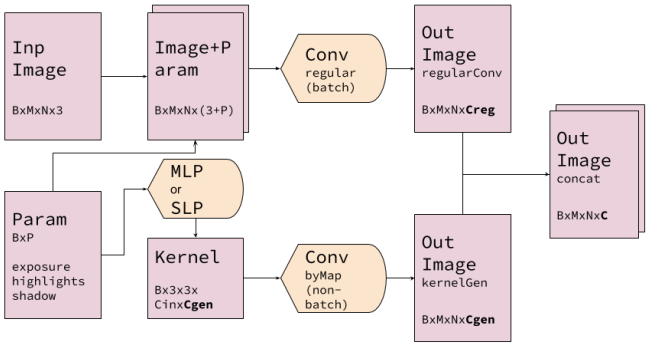


Figure 4: A hybrid kernel prediction layer

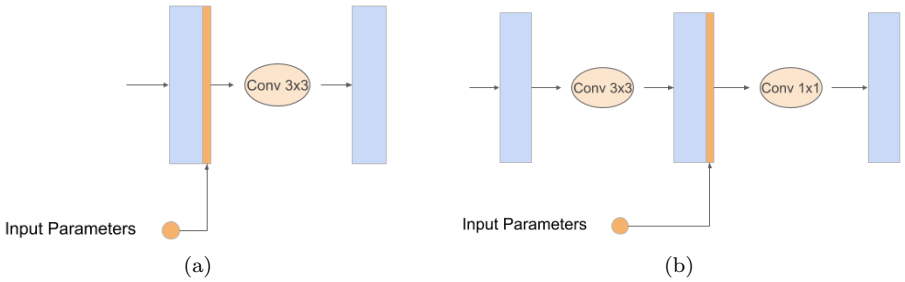


Figure 5: Comparison between common CNN blocks and parameter blocks

with parameter appended. This layer allows the parameter to interact with the local features extracted in the previous layers.

Note that parameter blocks reduce the number of parameters used in the model. If the block increase the number of channels in the activation map from C to $2C$, a regular 3×3 CNN layer will need $3 \times 3 \times C \times 2C = 18C^2$ parameters. Our parameter blocks only need $3 \times 3 \times C \times C + 1 \times 1 \times C \times 2C = 11C^2$ parameters.

4 Dataset

The experiments are based on the Adode 5K dataset[5]. We wrote a python driver for RawTherapee to take random parameters on a selected set of sliders. Our dataset are then grouped as triplets: an original image from Adoke 5K, a set of parameters used in RawTherapee to edit the image, and the output from RawTherapee that we want our model to predict.

The Adobe 5K dataset has 5,000 images. We apply two set of random

Table 1: Parameters used in RawTherapee. Parameters with very strong effect are selected with normal distribution ($\mathcal{N}$) to reduce the number of over- and under-exposed images which are rare after editing. All distribution are clipped by the effective range of the parameters.

Name	Min Value	Max Value	Random Selection
Exposure	-1	12	$\mathcal{N}(2, 2)$
Contrast	-100	100	$\mathcal{N}(0, 20)$
Highlights	0	100	$\mathcal{U}(0, 100)$
Shadows	0	100	$\mathcal{U}(0, 100)$

selected parameters on each images, resulting in 10,000 triplets. During training, we take 16 random patches from each triplet to speed up image processing and handle image size mismatch. See Table 1 for the parameters we used. It is simple to expand the parameter set and include more parameters desired.

Note that we are unable to use the editing professionals made in the original dataset. The parameters and software used in the professionals' editing is unknown. Also, we need general-purpose proxy functions that works for almost all combination of parameters. The professionals' choice may introduce a biased distribution.

5 Experiments

5.1 Single Layer Experiment

We illustrate the efficiency of kernel prediction model family with a simple task-fitting a proxy function for an "affine software". An affine software is an artificial image editing software that has only 2 parameters: scale k and bias b. Given an input image I, this software simply outputs $k \times I + b$ where the affine operations are applied on each pixel.

To include the input parameters in the receptive field of the convolution kernels, the input to the model is a concatenation of image and parameters (I, k, b). The parameters are appended as extra channels to the image. Each parameter value is repeated over the span of the image.

We will compare single-layer kernel prediction network with CNN of 1 and 2 layers. The single-layer kernel prediction will use its only convolution layer to predict a kernel $f_1(I, k, b)$ and a bias $f_2(I, k, b)$ to be applied on the input image. Note that f_1 and f_2 are affine transformations. The eventual output F_{KP} of this network is a quadratic function of input image. Meanwhile, 1-layer CNN is a linear function of fewer parameters. 2-layer CNN is a non-linear function with more parameters. (See Table 2).

$$F_{KP}(I, k, b) = f_1(I, k, b) * I + f_2(I, k, b)$$

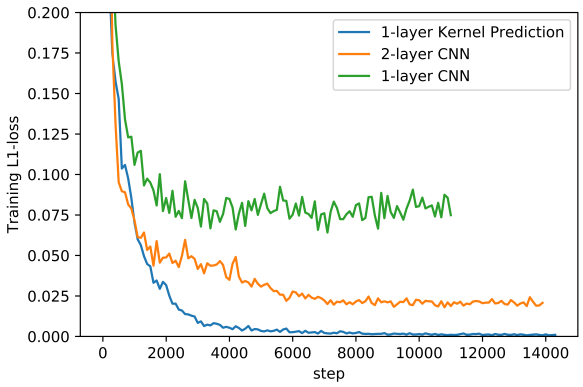


Figure 6: Single-layer kernel prediction network can converge faster and better than CNN of 1 or 2 layers

Table 2: Comparison of single layer models on affine software task (evaluation set)

Model	# of param	Function Type	L1-loss	L2-loss	PSNR
1-layer 1×1 CNN	18	Linear	0.07880	0.015738	18.03
2-layer 1×1 CNN	75	Non-linear	0.01910	0.000673	31.71
1-layer KPNet	72	Quadratic	**0.00148**	**0.000041**	**43.91**

Kernel prediction network can converge much faster to a lower loss on this task during training (See Figure 6). It also provides a much stronger performance on validation set in terms of L1- / L2-loss and PSNR value.

Kernel prediction network adopts a two-step approach where it first generates an operation based on the input image and parameters, and then apply this operation on the input image. In CNNs, the two steps are combined into one and the modifications to the image need to happen in-place. While modifying the image in-place is trivial for the kernel prediction network (you can write down the formula directly in this experiment), it is not clear that if a two-layer CNN can trivially fit the same thing. There is an intrinsic difficulty in linear layers or convolution layers to produce the product of two elements in a vector or cross channels.

5.2 Dummy Software Experiment

We further illustrate the capability of full-sized kernel prediction network on artificial datasets. We are not yet using the whole Adobe-5K dataset in this section. Instead, we use fixed noise image and fixed real images as the base of image generation. In comparison, we also used random noise images as the

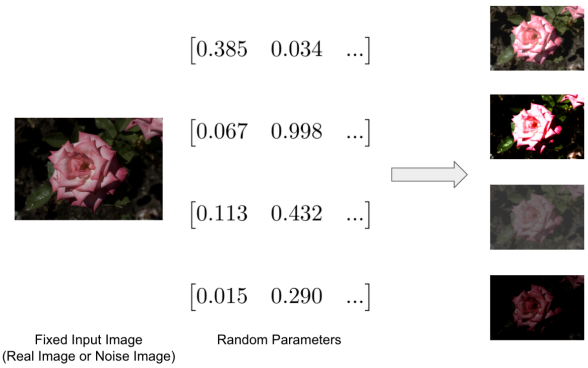


Figure 7: The data generation pipeline on fixed noise and real images.

base of image generation. Note that the parameters are still randomly selected. There are not duplicate triplets in the training or validation set.

Clipped affine software. This experiment uses a dataset generated with clipped affine software. The output of random affine transformation is clipped to fit within $(0, 1)$. This simulates real software's behavior as the pixel values in the output image never exceed 1 in float format. This also introduces extra challenges for the models than the experiments with only random affine transformation in the last section. Since the operation is purely pixel-based, the kernel prediction network only needs to predict one-by-one kernels.

The results can be found in Table 3. The kernel prediction network outperforms CNN-based U-Net on all 3 settings by nontrivial margin.

Note that the advantage of kernel prediction network on fixed images is much higher than those on random images. We believe that this shows that U-Net tends to overfit to the fixed images despite randomized input parameters. This also explains why U-Net improved greatly on random noise images. The kernel prediction network only applies an linear operation (i.e. a convolution) on the input image, which poses a strong **regularization**. This helps avoid over-fitting on restricted dataset.

Quadratic and bilateral filtering software. Our second experiment on the artificial dataset aims at demonstrating the capability of kernel prediction network on functions when local operations over multiple neighbouring pixels are required. The quadratic and bilateral filtering software takes in 2 parameters. The first parameter $p \in (0, 1)$ is used in the quadratic curve (visualized in Figure 9)

$$O = I + p \times I(1 - I)$$

The other two parameters σ_{color} and σ_{space} ($\sigma_{color}, \sigma_{space} \in (0, 200)$) are

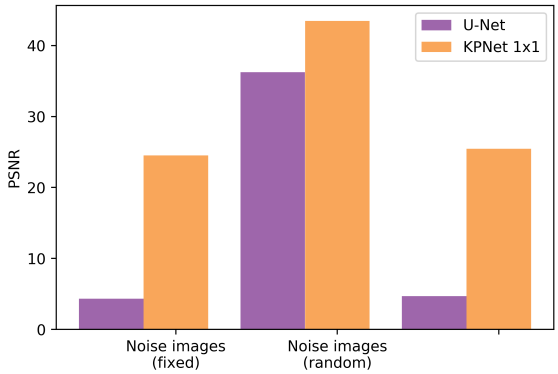


Figure 8: Comparison of models on different artificial tasks with affine and clipping operations.

Table 3: Comparison of U-net and kernel prediction network on artificial dataset with affine and clipping operations.

Training Mode	Model	L1-loss	L2-loss	PSNR
Noise images (fixed)	UNet	0.48776	0.36893	4.33
	KPNet 1×1	0.03406	0.00353	**24.51**
Real images (fixed)	UNet	0.45510	0.34006	4.68
	KPNet 1×1	0.03164	0.00285	**25.44**
Noise images (random)	UNet	0.00846	0.00023	36.24
	KPNet 1×1	0.00208	0.00004	**43.47**

used to parameterize a bilateral filter applied on the output of the quadratic function. This introduces local operations (5×5) to the output image, thus creating an opportunity to demonstrate kernel prediction network's ability to produce convolution kernels larger than 1×1.

Table 4 shows that kernel prediction network consistently outperform U-Net on a dataset which involves **local operations**. We believe this shows that it is more difficult for U-Net to fit a local operation while the kernel prediction network has this concept built into its second stage where convolutions are applied on the input image. The 3×3 kernel version also proves useful on random noise images.

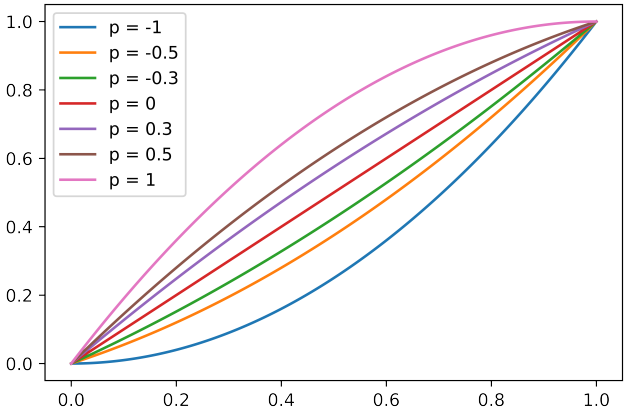


Figure 9: The effect of quadratic function.

5.3 Parameter Blocks

Now we will compare U-Net and kernel prediction network on all real images in our dataset processed dummy software. The software used here are clipped affine software and quadratic software.

We also include parameter blocks in the comparison. Recall that parameter blocks can theoretically achieve similar effect as simple convolution layers while using less parameters. It also avoids unnecessary linear correlation in the receptive field of each convolution kernel (e.g. for each 3×3 kernel, the same parameter appears 9 times).

Table 5 shows that kernel prediction network can consistently outperform U-Net in all settings by non-trivial margin. Parameter blocks can always improve the model performance compared to that without parameter blocks. Overall, the combination of kernel prediction and parameter blocks are the best on each dataset. Note that parameter blocks actually reduce the number of used parameters.

5.4 Real Data Experiment

On the real dataset where RawTherapee is used to process Adobe-5K dataset, kernel prediction network outperforms U-Net in all settings by non-trivial margin (Table 6). Meanwhile, parameter blocks prove useful as it not only helps kernel prediction network but also greatly improves U-Net.

6 Conclusion

We demonstrate a series of experiments to fit a proxy function for image editing software RawTherapee. We show that kernel prediction network outperforms

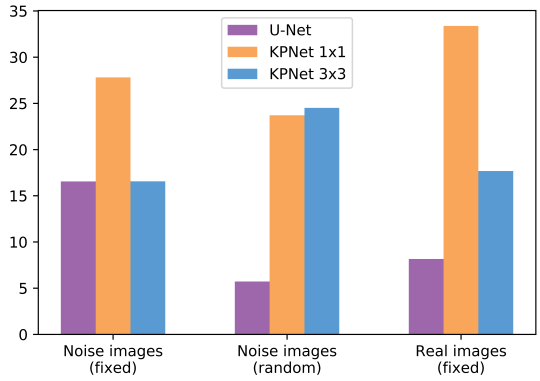


Figure 10: Comparison of models on different artificial tasks with quadratic and bilateral filtering operations.

U-Net as a proxy function. We also proposed parameter blocks which greatly improve model performance while using less parameters. Our studies on artificial dataset suggests that kernel prediction network are better suited for image editing due to its two-step approach-predicting the operations to be applied on the input image. This serves as a strong but helpful regularization on model that improves performance. Overall, the result of our experiments marks a critical step towards developing a deep learning model for non-destructive transfer between editing software.

References

[1] B. Mildenhall, J. T. Barron, J. Chen, D. Sharlet, R. Ng, and R. Carroll, "Burst denoising with kernel prediction networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2502–2510, 2018. 2, 4, 5, 6

[2] Y. Hu, H. He, C. Xu, B. Wang, and S. Lin, "Exposure: A white-box photo post-processing framework," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 2, p. 26, 2018. 4, 15

[3] T. Marinč, V. Srinivasan, S. Gül, C. Hellge, and W. Samek, "Multi-kernel prediction networks for denoising of burst images," in *2019 IEEE International Conference on Image Processing (ICIP)*, pp. 2404–2408, IEEE, 2019. 5

[4] A. Ignatov, L. Van Gool, and R. Timofte, "Replacing mobile camera isp with a single deep learning model," *arXiv preprint arXiv:2002.05509*, 2020. 5

Table 4: Comparison of U-net and kernel prediction network on artificial dataset with quadratic and bilateral filtering operations.

Training Mode	Model	L1-loss	L2-loss	PSNR
Noise images (fixed)	UNet	0.12352	0.02209	16.55
	KPNet 1×1	0.03195	0.00165	**27.81**
	KPNet 3×3	0.10420	0.02203	16.56
Real images (fixed)	UNet	0.32727	0.15267	8.16
	KPNet 1×1	0.01424	0.00045	**33.37**
	KPNet 3×3	0.08830	0.01702	17.68
Noise images (random)	UNet	0.49405	0.26798	5.71
	KPNet 1×1	0.05036	0.00425	23.71
	KPNet 3×3	0.04602	0.00353	**24.51**

[5] V. Bychkovsky, S. Paris, E. Chan, and F. Durand, "Learning photographic global tonal adjustment with a database of input / output image pairs," in *The Twenty-Fourth IEEE Conference on Computer Vision and Pattern Recognition*, 2011. 7

7 Supplementary: What is LightRoom/RawTherapee doing?

This part of experiments aim at understanding the underlying behavior of LightRoom/RawTherapee. The results provide insights and inspiration on what needs to be done and what could be done to solve our problem. We fitted some curve and model, but they are not necessarily part of our final solution.

7.1 What happens in those sliders

If the behavior of sliders in editing software are well-defined, we might even be able to produce close-form transformation from one software to another. However, we will reveal in this chapter that commercial editing software can be mysterious and convoluted:

- They may not reveal their algorithm (e.g. LightRoom is not open-sourced)
- Operations may be content-aware/semantic-dependent, or even just a little bit harder than naive algorithm so that it is impossible to reverse-engineer. (e.g. Adobe revealed that their operations are an "interpolation of different channel")

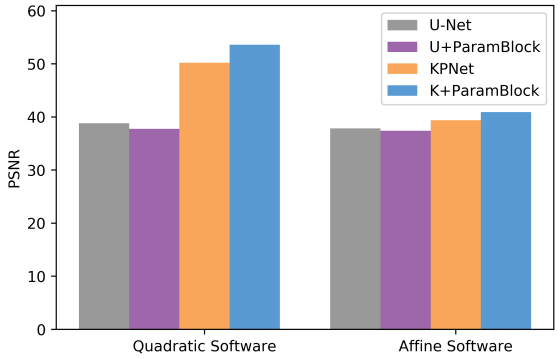


Figure 11: The effectiveness of Parameter Blocks

Table 5: Comparison of U-net and kernel prediction network on real dataset with quadratic and bilateral filtering operations.

Software	Model	ParamBlock	L1-loss	L2-loss	PSNR
Clipped Affine Software	KPNet	yes	0.00392	0.000080	**40.93**
	KPNet	no	0.00432	0.000115	39.37
	UNet	yes	0.00762	0.000181	37.41
	UNet	no	0.00676	0.000160	37.95
Quadratic Software	KPNet	yes	0.00102	0.000004	**53.60**
	KPNet	no	0.00166	0.000009	50.21
	UNet	yes	0.00787	0.000166	37.78
	UNet	no	0.00697	0.000131	38.81

- Software use image processing pipeline. (e.g. fixing pixel intensity after operations) This increases the entanglement between operations.

LightRoom is not an open-source software. So it's algorithms remain mysterious. We investigated some sliders of LightRoom: exposure, contrast, highlights shadow and white. Some of them behave as expected, others, not quite so. We will start with good-looking ones to the opposite ones.

Contrast. The white-box exposure paper [2] defines contrast as such

$$Lum = R * 0.27 + G * 0.67 + B * 0.06 \tag{1}$$

$$Lum_{enhanced} = 0.5 * (1 - \cos(\pi * Lum)) \tag{2}$$

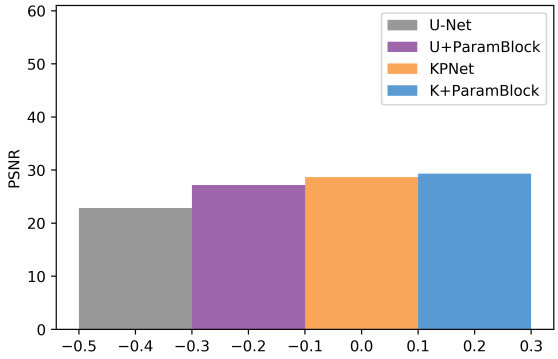


Figure 12: The performance of kernel prediction network on real images and RawTherapee.

Table 6: Comparison of U-net and kernel prediction network on real dataset on RawTherapee dataset.

Model	ParamBlock	L1-loss	L2-loss	PSNR
KPNet3 × 3	yes	0.0198	0.00117	**29.32**
KPNet3 × 3	no	0.0224	0.00135	28.67
UNet	yes	0.0270	0.00190	27.18
UNet	no	0.0432	0.00516	22.86

$$Image_{enhanced} = Image * (Lum_{enhanced}/(Lum + 1e - 6)) \tag{3}$$

$$Output = (1 - p) * Image + p * Image_{enhanced} \tag{4}$$

where p is half of the contrast slider value. This turns out to be reasonably close to skimage.contrast or LightRoom 13. Note that LightRoom histogram is smoother, unlike the naive implementation. This is intuitive while it also suggests that LR processes images in a pipeline where there could be multiple correlated steps. In general, this is an operation that we can almost understand.

Highlights. It is observed that in LR, pixel intensity in each channel increased linearly when highlights changes (see Figure 14a). This can also be interpreted as the linear combination scheme as above - editing algorithm defines the result when slider is moved to the end, and the output is an linear combination of input image and the output of the algorithm. Different original value leads to

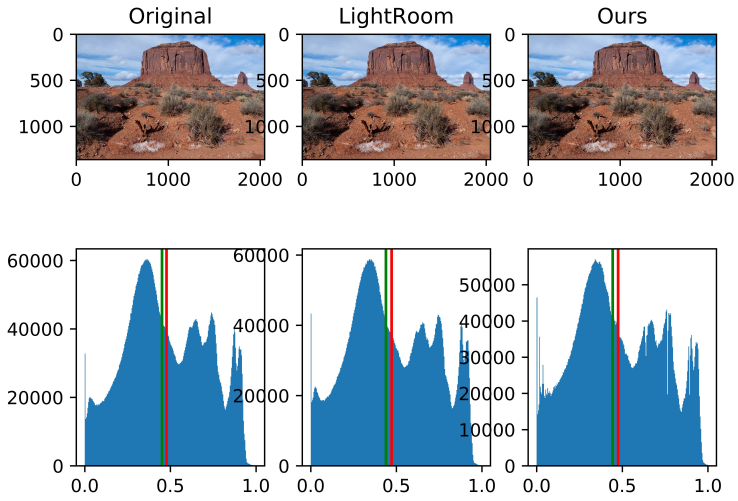


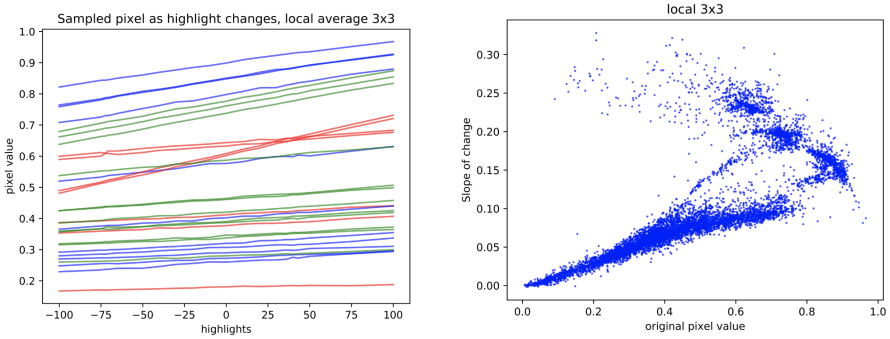
Figure 13: Comparison between LightRoom result and our re-implementation of contrast algo

different slope. Generally, higher slope is given to pixel with higher original value as the algorithm is supposed to highlight brighter pixels. We can also spot a few outliers. Note that this is not caused by channel-specific algorithm because the red trace in the middle surpass some other red trace at when highlight reaches 25. This means highlight operation **swap the order of pixel in the same channel**.

To illustrate the distribution of slope applied on each pixel, figure 14 shows the relation between slope (y axis) and original pixel intensity (x axis). There is a non-trivial cluster of outliers that is picked by LightRoom to have very high slope despite their original value. This leads to a lot of swap-of-order in pixel intensity.

Exposure. The "exposure" operation of LightRoom is the most studied operation till now - we tried to reverse-engineer and simulate it. The textbook formula is $O = I \times (2^{exposure})$ where the fact is obviously different, which will not be smooth where LR always generate smooth histogram and seldom collapse to maximum/minimum value when exposure is set to extreme.

Fit the curve with general logistic function. The "curve of exposure", a.k.a. response in pixel intensity when exposure slider is moved, looks like logistic functions (see figure 15a). This function should be parameterized by



(a) Effect of highlight slider on each pixel

(b) Slope on different pixel

Figure 14: The effect of the highlight operation is linear on each pixel.

exposure value x and original pixel intensity v. By assuming it is some logistic function, we can fit it with Bayesian optimization given some samples. The result is

$$ExposureCurve(x, v) = \frac{1}{1 + (2.59 - 2.456v)2^{-x}}$$

with an average difference of 0.1 (out of 1) on each element of the curve. This optimization is reliable as the loss shows a shallow low plateau in figure 15b. This formula can be further improved by introducing more terms (inspired by complexity of exposure in RawTherapee). It is not hard to fit some curve with reasonable loss, but there is no good explanation or intuition on explaining them. The function below gives a loss of 0.005 and select $c_1 = 0.9, c_2 > 3, c_3 = -0.161$ as an optimal after a few attempts.

$$\begin{aligned} f &= \frac{1}{1 + (k_1 v + b)e^{-x}} \\ g &= v2^{k_2 x} \\ output &= c_1 f + c_2 g + c_3 \end{aligned}$$

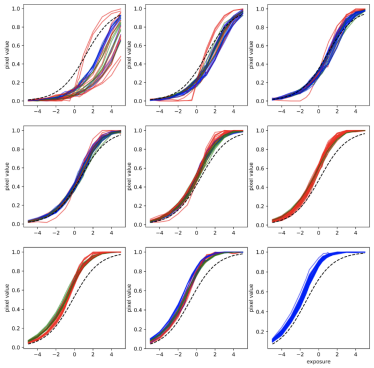
At the same time, we learnt that exposure is not a injective mapping: same pixel intensity ends up differently (confirmed by Adobe). 15c illustrates the mapping from original pixel intensity (x axis) and output pixel intensity (y axis) when exposure is set to 1.

Fitting the curve using DNN. Out of curiosity, we tried to use DNN to fit this curve. This experiment assume the curve is affected only by the three channel of a single pixel, i.e. it is entirely local. There is only one input: original pixel intensity of each channel, one pixel at a time. The network is supposed to

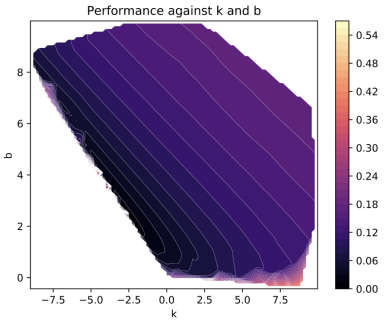
output curve of exposure for RGB channel separately. The naive dataset (1M rows of data) are some curves of exposure extracted from the same image, so this is not a generalized result. 15d shows some sample output.

7.2 RawTherapee's teaching

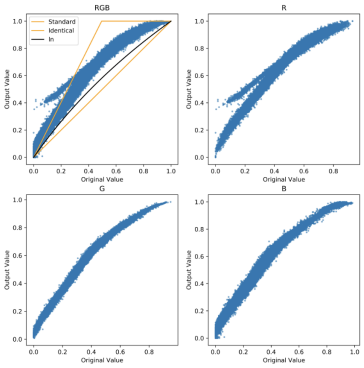
RawTherapee is open-sourced so we can watch how it handles different operations. We mostly studied how RawTherapee applies exposure on pictures. To make things more complex, RawTherapee introduces highlight compression, another slider, which works closely with exposure. (That being said, we don't



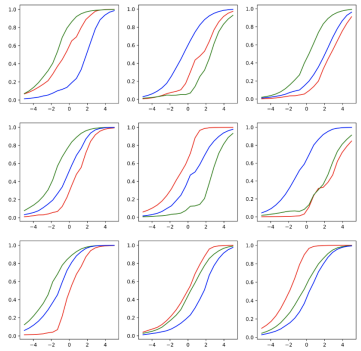
(a) Effect of exposure of pixel intensity (grouped by original value) / a.k.a. exposure curve



(b) Loss heatmap of Bayesian optimization on finding Logistic exposure curve



(c) Exposure is not an injective mapping in different channels



(d) Sample result by using DNN to fit exposure curve for single pixel

Figure 15: Exposure looks like logistic function so we can fit it though we know it is not an injective mapping

know how LightRoom sliders are entangled.) Some findings are

1. If highlight compression is 0, LightRoom are happy to use the textbook formula. If anything overflow (> 255), it just clips the value by 65535.
2. If highlight is not 0, it uses "the first curve". If anything overflow, it uses "the second curve". We can mock the curve well, at least for some parameters we have experimented with.
3. After applying the curve, there are other undiscovered process to clip the image.
4. It is hard to get intermediate result.
5. RawTherapee seems to be working on single pixel, but it is also observed through debugging tools that image size is in constant change although there is only one image in the workspace. Beside generating thumbnail, RawTherapee may also be scaling images or working on image patches.

The formula for exposure we learned from RawTherapee is (other parameters are removed for simplicity)

$$\begin{cases} \frac{p}{x}\log(1+pvx2^x), & x > 0 \\ v2^x, & x < 0 \end{cases} \tag{5}$$

That being said, RawTherapee result are significantly different from LightRoom, as mentioned in our problem statement.