Lightening Network for Low-light Image Enhancement: A JAX-based implementation

Mateus G Machado[1]

[1]*Centro de informática - Universidade Federal de Pernambuco*

June 13, 2024

Abstract

This report addresses the challenge of reimplementing the "Lightening Network for Low-light Image Enhancement" in JAX. The original work, featuring the "Deep Lightening Network" (DLN), presents a model that surpasses state-of-the-art low-light enhancement models. I undertook the task of translating the PyTorch code into JAX and comparing the results reported in the paper with those obtained using a model provided by the authors, a re-trained PyTorch model, and the JAX model. Metric results and image comparisons indicate that my reimplementation was successful, reducing the number of parameters by 280k and decreasing inference time by nearly 1 ms.

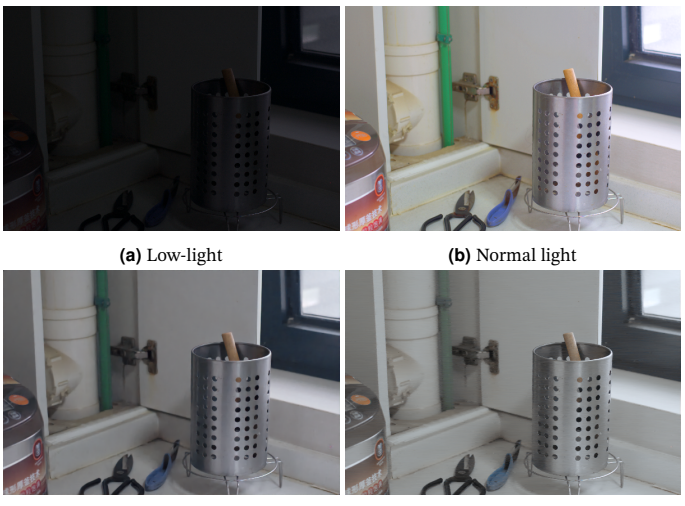
Keywords: *Enlightening model, Deep Lightening Network, Low-light Enhancement*

E-mail address: mgm4@cin.ufpe.br

1. Introduction

Capturing a high-quality photograph often involves overcoming several challenges, from achieving the correct focus to positioning for optimal lighting. However, it is not always possible to take photographs under perfect conditions. One of the most common issues is low-light conditions, where traditional solutions such as flash, ISO adjustment, and exposure time often have their own drawbacks.

To address this issue, [7] proposed the **Deep Lightening Network (DLN)**. Based on the concept that a normal light image can be modeled as a low-light image plus some residual component, they designed several residual blocks within the neural network architecture. The results reported in the paper are noteworthy, achieving a mean **Structure Similarity Index (SSIM)** [1] of **91.2%** on the LOL dataset [5], an **inference time** of **6.53 ms**, and comprising **700k parameters**. See in Figure 1 a sample result.



(a) Low-light **(b)** Normal light

(c) Authors prediction **(d)** JAX prediction

Figure 1. Visual comparison between predictions of the authors' provided fine-tuned network and my fine-tuned JAX network on a sample image from the LOL dataset.

In this report, I aim to:

1. Summarize the DLN architecture;
2. Re-implement the DLN using the JAX framework [3];
3. Detail the unreported aspects of the authors' implementation;
4. Evaluate the provided pre-trained and fine-tuned networks;
5. Retrain a DLN using the authors' source code [1];
6. Compare the authors' and retrained PyTorch [6] networks to my implementation quantitatively and qualitatively.

Finally, I present the performance metrics as reported in Table IV of the original paper, with the exception of Model Size due to issues encountered during the JAX implementation.

2. Methodology

2.1. Residual Learning

The DLN architecture assumes that low-light enhancement is a residual learning task. This means that a **normal-light (NL)** image $Y \in \mathbb{R}^{H\times W\times 3}$ can be reconstructed from a **low-light (LL)** image $X \in \mathbb{R}^{H\times W\times 3}$ when summed with an enhancing operator $P(\cdot)$. Mathematically,

$$Y = X + \gamma P(X) - n \tag{1}$$

where $n \in \mathbb{R}^{H\times W\times 3}$ represents the noise to be removed (ignored by [7]), and $\gamma \in \mathbb{R}$ is an interactive factor that controls the lightening power of the low-light enhancement (introduced by [7]). The enhancing operator $P(\cdot)$ is a learned function using a **Convolutional Neural Network (CNN)** structure, which the authors called DLN. The authors point out in the document that γ is extremely valuable to the final result, but, as I discuss later, they did not use this term in the equation.

2.2. Deep Lightening Network (DLN)

As the authors consistently highlight, the DLN architecture is based on residual learning. The inner blocks of the architecture also use this idea. They subdivided the structure into: **Feature Extraction**, **Lightening Back-Projection (LBP)** blocks, **Feature Aggregation (FA)** blocks, and the **Residual Enlightening Process**. Figure 2 provides a visualization of the general process of a DLN architecture.

2.3. Lighten Back-Projection (LBP)

The LBP blocks' architecture is inherited from the Back Projection technique, described in [4]. It leverages two sequential Encoder-Decoder architectures in each block to utilize their residuals. See

[1] https://github.com/WangLiwen1994/DLN

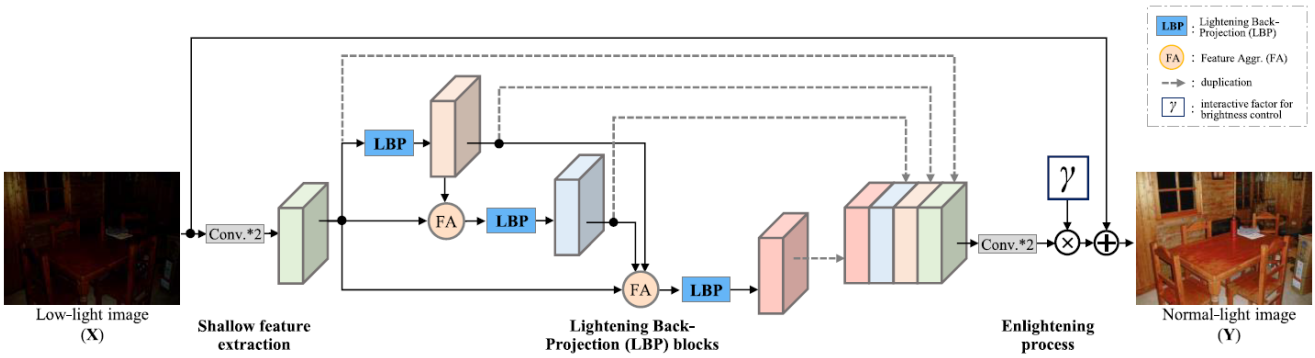


Figure 2. Architecture of the Deep Lighten Network (DLN). Source [7].

Figure 3 for the core idea of the LBP block. Note that the L_1 operation functions as the encoder for the D operation, which serves as the decoder. Subsequently, the D operation acts as the encoder for the L_2 operation, which is the final decoder.

We can describe the LBP such that, given a LL image ($\mathbf{X}$), an NL image ($\mathbf{Y}$) can be obtained following the equation:

$$\hat{Y} = \lambda_2 L_1(X) + L_2(D(L_1(X)) - \lambda_1 X), \tag{2}$$

where $\lambda_1 \in \mathbb{R}$ and $\lambda_2 \in \mathbb{R}$ are weights to balance the residual updating, L_n are the Lightening operations, and D is the Darkening operation. Ideally, the Darkening operation output ($\widetilde{X}$) is the same as the ground-truth (X). In real conditions, the residual ($R_{LL} = X - \widetilde{X}$ with $R_{LL} \in \mathbb{R}^{H\times W\times 3}$) indicates the weakness of the lightening (L_1) and Darkening (D) operations. This information allows for estimating the residual in the NL domain ($\widetilde{R}_{NL} \in \mathbb{R}^{H\times W\times 3}$, for $\widetilde{R}_{NL} \approx Y - \widetilde{Y}$) through a second lightening operation (L_2). Ultimately, $\hat{Y} \in \mathbb{R}^{H\times W\times 3}$, the refined NL image, can be estimated by adding $\widetilde{R}_{NL}$ to $\widetilde{Y}$, i.e., $\hat{Y} = \widetilde{Y} + \widetilde{R}_{NL}$.

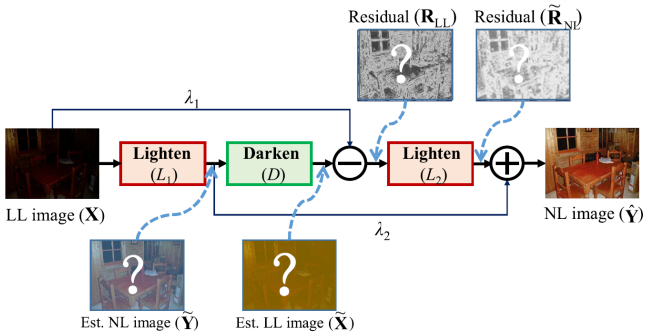


Figure 3. General architecture of a Lighten Back-Projection block. Source: [7].

2.3.1. Inner Operations

The Lightening and Darkening operations are the key components of the LBP blocks. Each follows an Encoder-Decoder architecture with a residual learning approach. These operations are performed in three steps:

1. **Encoding**: Responsible for feature extraction.
2. **Offset Estimating**: Responsible for learning the differences between the NL and LL images.
3. **Decoding**: Responsible for reconstructing the input, either enlightened or darkened, depending on the operation type.

In Lightening operations, the estimated offset is added to the encoded features, whereas in Darkening operations, it is subtracted. Both operations can be described as:

$$\begin{aligned} L(X) &= Dec(Enc(X) + Offset(Enc(X))), \\ D(\widetilde{Y}) &= Dec(Enc(\widetilde{Y}) - Offset(Enc(\widetilde{Y}))), \end{aligned} \tag{3}$$

where Enc, Offset, and Dec are the encoding, offset estimating, and decoding operations, respectively.

2.4. Feature Aggregation

The **Feature Aggregation (FA)** blocks are responsible for constructing meaningful feature maps to serve as inputs for the LBP blocks. See Figure 4 for a representation of the second FA block depicted in Figure 2.

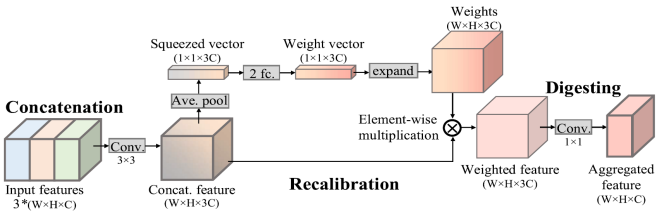


Figure 4. Structure of a three-input Feature Aggregation block. Note that the aggregated feature is a feature map with the same size as the original image $W \times H \times C$. Source: [7].

The FA consists of three parts:

- **Feature Concatenation**: Concatenating the N inputs along the channel axis. For example, three inputs of size $W \times H \times C$ result in $W \times H \times 3C$.
- **Recalibration**: Responsible for weighing each concatenated feature map. The idea is to learn a function that maps the average values of the features to weights. These weights are then element-wise multiplied by the concatenated features.
- **Digesting**: Responsible for processing the information generated by the recalibration operation and transforming it into useful features for the input to the LBP block.

2.5. Loss Function

The authors chose to define the DLN loss according to two measures: Structural Similarity and the **Total Variation (TV)** of the resulting image. The **Structural Similarity Index (SSIM)** is a well-known metric for assessing the quality of an estimated image [1]. SSIM can be defined as:

$$SSIM(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1} \cdot \frac{2\sigma_{xy} + c_2}{\sigma_x^2 + \sigma_y^2 + c_2} \tag{4}$$

where $x \in \mathbb{R}^{H\times W\times 3}$ and $y \in \mathbb{R}^{H\times W\times 3}$ are the images to be measured. The terms $\mu_x, \mu_y \in \mathbb{R}$ are the means, and $\sigma_x, \sigma_y \in \mathbb{R}$ are the variances

of the respective images. The constants $c_1 \in \mathbb{R}$ and $c_2 \in \mathbb{R}$ exist to prevent division by zero. Since SSIM ranges from 0 to 1, the structural similarity loss can be defined as $Loss_{struct}(\hat{Y}, Y) = 1 - SSIM(\hat{Y}, Y)$.

The TV loss is based on the idea that a light focus is not just a pixel or point in the image. When an image has a light focus, it will have a certain radius dispersing the light until it reaches darkness. The TV loss accounts for this smooth transition by minimizing the gradient of the entire image. The $Loss_{TV}$ can be described as:

$$Loss_{TV}(\mathbf{P}) = \sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{k=1}^{3}\left(\frac{(p_{i,j,k} - p_{i+1,j,k})^2}{3H(W-1)} + \frac{(p_{i,j,k} - p_{i,j+1,k})^2}{3(H-1)W}\right) \quad (5)$$

where $\mathbf{P} \in \mathbb{R}^{H\times W\times 3}$ represents the image to be measured, and $p \in \mathbb{R}$ is the value of the pixel indexed by i, j, k. The whole Loss equation is described as $Loss_{dif}(\hat{Y}, Y) = Loss_{struct}(\hat{Y}, Y) + \lambda Loss_{TV}(\hat{Y})$, where $\lambda \in \mathbb{R}$ is is the balance coefficient.

3. Implementation

3.1. JAX

JAX, short for "Just Another eXperimental framework," is a library developed by Google primarily for high-performance numerical computing, particularly for machine learning and scientific computing tasks [3]. It is an extremely powerful numpy-based tool which can natively run parallel hardwares such as GPUs and TPUs. JAX is not impactful in the research community as PyTorch (the framework used in the original code) or Tensorflow2 due to its functional programming paradigm and not-so-user-friendly code style.

```python
import jax
import jax.numpy as jnp
from jax import grad, jit, random

# Define the neural network architecture
def init_params(rng, input_size, hidden_size, output_size):
    """Initialize the parameters of the neural network."""
    w_key, b_key = random.split(rng)
    w1 = random.normal(w_key, (input_size, hidden_size))
    b1 = jnp.zeros((hidden_size,))
    w2 = random.normal(w_key, (hidden_size, output_size))
    b2 = jnp.zeros((output_size,))
    return (w1, b1, w2, b2)

def forward(params, inputs):
    """Forward pass of the neural network."""
    w1, b1, w2, b2 = params
    hidden = jnp.tanh(jnp.dot(inputs, w1) + b1)
    return jnp.dot(hidden, w2) + b2

# Define the loss function
def loss(params, inputs, targets):
    """Compute the cross-entropy loss."""
    preds = forward(params, inputs)
    return -jnp.mean(jax.nn.log_softmax(preds) * targets)

# Define the update function using gradient descent
@jit
def update(params, inputs, targets, lr=0.1):
    """Update the parameters using gradient descent."""
    grads = grad(loss)(params, inputs, targets)
    return [(param - lr * grad_param) for param, grad_param in zip(params, grads)]

model_params = init_params(rng, input_size, hidden_size, output_size)
```

Code 1. Simple Neural Network using JAX

The pure JAX code for creating and training a simple neural network reminds me a lot the code-style of Tensorflow1. For this, the community brought FLAX alive, end-to-end and flexible API written using JAX [9]. Codes 1 and 2 showcases the simplified and friendly code of FLAX, from a clear way to define a model class to the training step. In addition, FLAX just made available the NNX ("Neural Network library for JAX") to acquire more JAX users, with a PyTorch-like API. I have developed the DLN in both FLAX and FLAX-NNX methodologies, but as NNX is recent, I did not find examples of saving the model or the model's parameters[2].

```python
from flax import linen as nn
import jax
import jax.numpy as jnp
from jax import random

# Define the neural network architecture using Flax
class SimpleDNN(nn.Module):
    hidden_size: int
    output_size: int

    @nn.compact
    def __call__(self, x):
        x = nn.Dense(self.hidden_size)(x)
        x = nn.relu(x)
        x = nn.Dense(self.output_size)(x)
        return x

# Define the loss function
def cross_entropy_loss(logits, labels):
    return -jnp.mean(jax.nn.log_softmax(logits) * labels)

# Define the training step
@jax.jit
def train_step(optimizer, batch):
    def loss_fn(model):
        logits = model(batch["image"])
        loss = cross_entropy_loss(logits, batch["label"])
        return loss
    grad_fn = jax.value_and_grad(loss_fn)
    loss, grad = grad_fn(optimizer.target)
    optimizer = optimizer.apply_gradient(grad)
    return optimizer, loss

# Initialize model and optimizer
model = SimpleDNN(hidden_size=hidden_size, output_size=output_size)
optimizer = jax.opt.Adam(learning_rate=0.1).create(model)
```

Code 2. Simple Neural Network using FLAX

3.2. Translating Losses and Metrics

As detailed in Sections 2 and 4, the original framework relied on Total Variation (TV) and Structural Similarity Index (SSIM) as loss functions, alongside SSIM and **Peak Signal-to-Noise Ratio (PSNR)** as performance metrics. Initially implemented with scikit-image [2], these metrics were integral to the training and evaluation phases. However, when transitioning to JAX, it became necessary to adapt these losses and metrics.

For this purpose, I employed DeepMind's PIX, a JAX-based image processing library [3], to compute SSIM and PSNR, while also crafting my implementation of the TV loss. In the notebook titled "compare_metrics.ipynb", I conducted a thorough comparison between the results obtained from scikit-image's and PIX's implementations. The findings, summarized in Table 1, revealed a remarkable consistency between both frameworks, affirming their suitability for utilization in the JAX environment.

[2] My repository is available in https://github.com/goncamateus/DLN-jax

Table 1. Losses and metrics comparisson from the original and JAX implementations.

	Original	JAX
SSIM	0.093	0.098
TV	0.009	0.009
PSNR	13.263	13.263

3.3. Paper vs Code

Here I point out where the provided code by the authors differs from the paper description[3]. In a general list, the authors did not report:

1. a feature concatenation;
2. why they did not use the iterative γ factor;
3. bias initialization;
4. reverse residual subtraction.

The forward method of the DLN model class is defined as:

```
1  def forward(self, x_ori, tar=None):
2          # data gate
3          x = (x_ori - 0.5) * 2
4          x_bright, _ = torch.max(x_ori, dim=1,
       keepdim=True)
5          x_in = torch.cat((x, x_bright), 1)
6
7          # feature extraction
8          feature = self.feat1(x_in)
9          .
10         .
11         .
12         feature_out = self.feature(feature_in)
13         pred = self.out(feature_out) + x_ori
14         return pred
```

3.3.1. Feature Concatenation

Lines 3 to 5 provide information that was not reported. The inputs of the model X are not in $\mathbb{R}^{H\times W\times 3}$, but in $\mathbb{R}^{H\times W\times 4}$. Line 4 of the method retrieves the maximum value along the channel axis, with $X_{bright} \in \mathbb{R}^{1\times 1\times C}$. Further, they concatenate this feature to the image. This process is not shown anywhere in the paper. In fact, there is an open (and unanswered) issue on their repository [4].

3.3.2. Iterative factor

As mentioned in the introductory section:

> "We resolve the low-light enhancement through a residual learning model that estimates the residual between the low- and normal-light images. The model has an interactive factor that controls the power of the low-light enhancement." [7]

Except that they did not use it in this code. The γ factor should be used in line 13, as follows:

```
1  pred = gamma * self.out(feature_out) + x_ori
```

I did not try using it, as they explain that γ is a non-linear learned function, and I had no time to reproduce that.

3.3.3. Minor Problems

Three minor problems I have found in the original implementation are the bias initialization with zeros of the DLN model and submodels, the inconsistent number of training epochs for both datasets, and the reverse residual subtraction on LBP.

The document states that, when pre-training the model, they used 100 epochs and they do not specify how many epochs they trained for fine-tuning. In the code, I understood that they actually trained for 500 epochs in both settings. When I tested using only 100 epochs to pre-train, the results were too different from the reported. For that **I used 500 epochs in both training settings**.

It is clear in the paper that $\widehat{Y} = \lambda_2 L_1(X) + L_2(D(L_1(X)) - \lambda_1 X)$, but in the code it is $\widehat{Y} = \lambda_2 L_1(X) + L_2(\lambda_1 X - D(L_1(X)))$ [5]. Clearly, during the learning process, the model learns in both ways. Just for checking, I ran the code in both ways and the models achieved the same performance.

4. Experiments

4.1. Datasets and Data Loading

To ensure a fair comparison of results, I utilized the same datasets as the authors. The VOC2007 dataset [10] was employed for pre-training the model, while the Low-Light (LOL) dataset [5] was used for fine-tuning. Both datasets were sourced from their original websites and augmented as indicated in the code. It is important to note that if the authors used a different version of the LOL dataset, it should have been highlighted in their paper.

Since neither JAX nor FLAX provide a native data loader class (as found in PyTorch and TensorFlow2), I had to adapt the final procedure of the original data loader used by the authors. I created the data loader method using a Python generator [8], with each iteration yielding the transposed JAX-numpy array of the image, already pre-processed and augmented.

See Code 3 for the method to obtain both LL and NL images. Note that I had to transpose both images, as PyTorch's CNN-2D processes the signal in the form $X \in \mathbb{R}^{3\times H\times W}$, while FLAX uses the format $X \in \mathbb{R}^{H\times W\times 3}$.

```
1  def get(self, shuffle=True):
2          indices = np.arange(len(self.dataset))
3          if shuffle:
4              np.random.shuffle(indices)
5          for start_idx in range(
6              0, len(self.dataset) - self.
       batch_size + 1, self.batch_size
7          ):
8              excerpt = indices[start_idx :
       start_idx + self.batch_size]
9              batch = [self.dataset[i] for i in
       excerpt]
10             low_light, normal_light = zip(*batch
       )
11             jaxed_low_light = jnp.array(
       low_light)
12             jaxed_normal_light = jnp.array(
       normal_light)
13             yield jnp.transpose(jaxed_low_light,
        (0, 2, 3, 1)), jnp.transpose(
14                 jaxed_normal_light, (0, 2, 3, 1)
15             )
```

Code 3. Data acquisition method adaptation for JAX

To analyze the results of the pre-trained model on the VOC2007 dataset, I used the first 15 photos of the test set. These 15 images were neither patched nor augmented; instead, I followed the same process to create synthetic low-light (LL) photos as described by the authors. This process is detailed in the notebook `generate_voc2007_ll.ipynb`.

4.2. Results

For the results, I will present a quantitative comparison of four models using specified metrics, and a qualitative analysis for three models. The reports are structured as follows:

- **Paper**: Results reported in the original paper.

[3] Disclaimer: I accessed the original repository on 03/19/2024, so it may be a different version

[4] https://github.com/WangLiwen1994/DLN/issues/2

[5] https://github.com/WangLiwen1994/DLN/blob/c4aca5237babfeaf6fd98117e0697218527655e4/model.py#L70

Table 2. Results of Paper, Original, Re-trained, and JAX moels. The metric "Parameters" if for the number of trainable parameters that the model has, "Time" is the mean inference time of a single image, and "θ" is the angular error between the NL image and the estimated NL image. "PSNR" and "SSIM" are the Peak Signal Noise Ratio and Structural Similarity, respectivelly. The angular error was not reported in the paper, so I can not report it either.

			VOC2007			LOL		
	Parameters	Time	PSNR	SSIM	θ	PSNR	SSIM	θ
Paper	700k	6.53 ms	23.82	0.91	-	21.94	0.80	-
Original	700k	3.67 ms	25.19	0.89	**2.52**	**21.94**	**0.84**	**3.67**
Re-trained	700k	3.59 ms	21.27	0.84	3.38	19.53	0.79	4.75
JAX	**421k**	**2.75 ms**	**27.77**	**0.90**	3.23	20.45	0.80	4.90

- **Original**: Results obtained using the pre-trained and fine-tuned models provided by the authors in their repository.
- **Re-trained**: Results of the model trained using the authors' original code.
- **JAX**: Results of the model trained using my reimplementation with the JAX/FLAX frameworks.

Unfortunately, by the time I completed this document, the authors had removed the download link from their repository. However, when I initially accessed it, I was able to download both the pre-trained and fine-tuned models.

4.3. Quantitative Comparison

As part of the Computer Vision course, the angular error was introduced as a performance measure for color correction, so I included this metric in my analysis. In the notebook `compare_results.ipynb`, I quantitatively compared the results of the original, re-trained, and JAX reimplemented models for both pre-trained and fine-tuned versions. Table 2 shows the numerical comparison of the four models.

Unfortunately, I could not compare the model sizes, as it is unclear how the authors measured them. It would be unfair to compare the size of a model optimally compressed to a JSON file (the current optimal way to save JAX models' weights).

Interestingly, the reimplemented JAX model has fewer parameters than the PyTorch models. This is a new finding, as I have not seen comparisons of this nature elsewhere. Directly related to the parameter size, the inference time of the JAX model is also lower than that of the others. The authors considered the first timing, which includes GPU allocation time, making their reported results potentially less accurate. The results presented in Table 2 are likely more accurate.

The PSNR, SSIM, and angular error metrics from all four models are comparable in both pre-trained and fine-tuned scenarios, demonstrating our success in reimplementing the original work. A study of 10 random initializations for pre-training and fine-tuning should be conducted for a more thorough analysis of the model results. However, since the authors did not conduct such a study, I chose not to either.

4.4. Qualitative Comparison

For the visual analysis, Figures 5 and 6 show the estimated normal-light (NL) images from the Original, Re-trained, and JAX models. All these images can be better observed in my repository in the folder `notebooks/results`.

In particular, the last image of Figure 6 highlights the difficulty for DLN models to accurately estimate the colors in the digital clock. A similar issue occurs in the second image of Figure 5, where the focus of light caused by the sun has a direct reflection on the camera's lens. This effect may be attributed to the introduction of the total variation (TV) loss, which aims for smooth light dissipation. While light dissipation generally occurs, these two images illustrate instances where the TV loss fails, emphasizing the importance of fine-tuning the λ balance coefficient of $Loss_{TV}$.

Overall, the Original model appears to provide visually better estimations of the images. The DLN models struggle to output non-noisy images. For instance, in Figure 5, the estimations seem to have a grey layer, and in Figure 6, the predominant color turns yellow. Applying Gray World or White Patch algorithms might improve the final results.

5. Conclusion

Recognizing the growing prominence of the JAX framework, I embarked on advancing my studies in computer vision using this powerful tool. As detailed above, I successfully reimplemented a deep learning model for color correction using JAX.

I also highlighted discrepancies between the PyTorch-like code provided by the authors and the specifics outlined in the original paper. It is important to note that I did not extensively explore parameterization or experiment with all features as detailed in the paper, unlike the original code. Nonetheless, the publicly available model performed well and yielded results similar to those reported.

Reimplementing this model provided valuable insights that will benefit my ongoing research in robotics. While initially daunting, adapting to JAX proved rewarding and will undoubtedly facilitate future model implementations or adaptations.

References

[1] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. DOI: 10.1109/TIP.2003.819861.

[2] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, *et al.*, "Scikit-image: Image processing in python," *PeerJ*, vol. 2, e453, 2014.

[3] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/google/jax.

[4] M. Haris, G. Shakhnarovich, and N. Ukita, *Deep back-projection networks for super-resolution*, 2018. arXiv: 1803.02735 [cs.CV].

[5] C. Wei, W. Wang, W. Yang, and J. Liu, *Deep retinex decomposition for low-light enhancement*, 2018. arXiv: 1808.04560 [cs.CV].

[6] A. Paszke, S. Gross, F. Massa, *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG].

[7] L.-W. Wang, Z.-S. Liu, W.-C. Siu, and D. P. K. Lun, "Lightening network for low-light image enhancement," *IEEE Transactions on Image Processing*, vol. 29, pp. 7984–7996, 2020. DOI: 10.1109/TIP.2020.3008396.

[8] L. Ramalho, *Fluent python*. " O'Reilly Media, Inc.", 2022.

[9] J. Heek, A. Levskaya, A. Oliver, *et al.*, *Flax: A neural network library and ecosystem for JAX*, version 0.8.5, 2023. [Online]. Available: http://github.com/google/flax.

[10] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results*, http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

(a) Low Light **(b)** Re-trained **(c)** Original **(d)** JAX **(e)** Ground Truth

Figure 5. Results using the synthetic dataset for testing the pre-trained models in VOC2007 dataset.

(a) Low Light **(b)** Re-trained **(c)** Original **(d)** JAX **(e)** Ground Truth

Figure 6. Results using the real for testing the fine-tuned models in LOL dataset.