Deep Learning for Synthetic Bokeh - Subject Separation Blur U-Net Study

Authors: Michael DB and others

Abstract

This project explores the application of deep learning to replicate bokeh effects synthetically, hoping to achieve the traditional results from high-aperture optics used in photography. Inspired by the PyNET Model, we explore the use of a custom U-Net based architecture, in hopes that we will be able to maintain the significantly strong performance with a 6.2x smaller, more efficient model. Training the model on a curated subset of the Everything is Better with Bokeh! (EBB!) dataset, our model runs a four-level encoder-decoder structure with skip connections and a weighted perceptual loss as compared to their seven-level PyNET Model. We showcase superior performance in quantitative metrics such as PSNR and LPIPS compared to the baseline PyNET Model, whilst also examining the changes these metrics lead to in qualitative outputs. In addition, we highlight how these quantitative metrics are not good indicators necessarily for successful bokeh generation. Overall, we show how a 7.70M parameter model trained on limited hardware can exhibit strong performance on the photographic bokeh generation task even when compared to its PyNET 47.5M parameter counterpart.

1 Introduction

This paper presents a deep learning-based approach to render natural camera bokeh effects from images taken at lower aperture settings. Bokeh, characterized by a pleasing subject-background separation through dramatic background blur, is traditionally achieved with high-aperture optics in expensive camera lenses. Machine learning and deep learning models have seen significant interest in creating synthetic bokeh, particularly for applications on cheaper consumer hardware such as mobile phones. Significant opportunity exists to improve these models for photography software to allow better availability to the public to render this pleasing subject separation blur. Our methods aim to synthetically generate realistic bokeh effects by training an efficient and accurate U-Net model. We will be focusing on preserving foreground details while achieving a natural and aesthetically pleasing background blur.

2 Background

2.1 Literature Review

In our initial research for this project, the work "Rendering Natural Camera Bokeh Effect with Deep Learning" stood out significantly [2]. Their end-to-end model, PyNET, directly tackles the Bokeh generation problem and became a key inspiration for our project. It outperformed other methods in user studies and showed proficiency in handling borders between focused and blurred regions with exceptional realism. A major factor that enabled this approach was the EBB! dataset which consisted of a large collection of DSLR image pairs taken at wide and shallow apertures. This dataset allows models to learn the mapping from sharp to blurred directly, avoiding potentially error-prone intermediate steps like depth estimation.

What was particularly impressive was PyNET's multi-resolution pyramid structure, which processes the image at progressively higher scales and merges the information to synthesize a refined output. This mimics the way depth and focus change across spatial scales and allows the network to capture both global scene context and fine local details. In addition, they employed a carefully designed training strategy combining a sequential, level-by-level pre-training routine with a composite loss function incorporating L1, SSIM, and VGG perceptual losses. This reflects advanced training techniques discussed in the course, aiming to optimize for both pixel accuracy and perceptual quality. The result was SoTA realism, achieving output comparable or even preferred to those from the Google Pixel camera pipeline. Another compelling aspect was the model's practical ability and runtime performance. Achieving inference times under 5.5 seconds on high-end mobile GPUs (without instance normalization) showed that high-quality Bokeh synthesis was becoming feasible beyond offline rendering or powerful desktop GPUs.

While the PyNET approach demonstrated significant success in generating realistic Bokeh, its complexity and large parameter count (47.5M), poses significant barriers to deployment in a resource-limited environment. In order to navigate this we looked deeper into PyNET's multi-level, pyramid architecture which differs from a typical CNN made up of a stack of convolutional layers followed by pooling. As a result, our work drew focus on designing a lightweight CNN model while retaining competitive performance and utilising the PyNET approach as a benchmark. Our study aims to investigate how by adapting the CNN into a U-Net architecture that utilises model symmetry and skip connections, we can further explore the capabilities and limitations of generating synthetic bokeh.

2.2 Our Contributions

- A more efficient, 83.8% smaller model which performs better quantitatively than the 47.5M parameter PyNET model on the EBB! test data in two metrics: PSNR and LPIPS
- An insight into the shortcomings of quantitative evaluation in bokeh generation given PyNET's superior qualitative performance
- A qualitative insight into how tweaking the weights of the loss function with respects to SSIM and LPIPS can affect the final product of the created synthetic bokeh blur
- Limitations of training and evaluating our small CNN model on a complex task such as bokeh generation, compared to the larger 7-level approach of PyNET which yielded superior qualitative results

3 Data

3.1 Data Description

This project uses the dataset: Everything is Better with Bokeh! (EBB!) [2]. It consists of 10,000 images which consist of 5000 image pairs that differ only by the depth of field. The authors of the original PyNET paper kindly created this dataset, with two photos being taken for each scenario on their Canon 7D DSLR camera and 50mm f/1.8 lens, with the first image captured with a narrow aperture (f/16) and the second with their highest aperture (f/1.8) [2].

3.2 Dataset Preprocessing

When downloading the dataset, we noticed that the test set folder was incorrectly labelled as the validation. In actuality there were 200 test image pairs, and 4694 other image pairs - we opted for an 80:20 train-validation split on these 4694 pairs. Leaving us with 3 separate data subsets for training, validation, and testing.

For speed and efficiency for the architecture and hyperparameter optimisation experimentation, seen in Section 5.2, we employed a smaller subset of 1024 training pairs and 180 validation pairs. In the full training phase, we increased the dataset size beyond this to a subset size of 2048. However, we deliberately avoided using all 4,694 pairs due to the time and hardware constraints. This medium sized subset retained sufficient diversity for generalisation while keeping the training times feasible for us to run on our available hardware.

4 Methodology

4.1 Vanilla U-Net

In this project, we developed a custom convolutional neural network (CNN) through a U-Net Architecture to simulate bokeh (lens blur) effect in RGB photographs. Unlike a CNN, which typically lacks symmetric expansion and contraction paths, U-Net is characterized by its encoder-decoder symmetry and skip connections. We refer to our model as Vanilla U-Net, hence the VBase naming convention used in our experiments in Section 5. The network is built to learn the transformation from an in-focus image to its corresponding shallow depth-of-field version.

4.2 Architecture

To effectively ensure that we were able to simplify the process of creating the bokeh as compared to the observed PyNET architecture, we decided that creating 4 levels for the CNN could reduce the model complexity, increasing training speed while reducing memory consumption to accommodate computing constraints.

Below is a demonstration of a forward pass for our model:

Algorithm 1 Forward Pass of `Vanilla U-Net`

Input: Sharp image tensor x
Output: Predicted bokeh image $\hat{y}$
procedure FORWARDPASS(x)
    $x_1 \leftarrow \text{enc1}(x)$
    $p_1 \leftarrow \text{MaxPool}(x_1)$
    $x_2 \leftarrow \text{enc2}(p_1)$
    $p_2 \leftarrow \text{MaxPool}(x_2)$
    $x_3 \leftarrow \text{enc3}(p_2)$
    $p_3 \leftarrow \text{MaxPool}(x_3)$
    $x_4 \leftarrow \text{enc4}(p_3)$ ▷ Deepest encoder level
    $u_3 \leftarrow \text{UpConv}(x_4)$
    $u_3 \leftarrow \text{PadToMatch}(u_3, x_3)$
    $cat_3 \leftarrow \text{Concat}(u_3, x_3)$
    $x_5 \leftarrow \text{dec3}(cat_3)$
    $u_2 \leftarrow \text{UpConv}(x_5)$
    $u_2 \leftarrow \text{PadToMatch}(u_2, x_2)$
    $cat_2 \leftarrow \text{Concat}(u_2, x_2)$
    $x_6 \leftarrow \text{dec2}(cat_2)$
    $u_1 \leftarrow \text{UpConv}(x_6)$
    $u_1 \leftarrow \text{PadToMatch}(u_1, x_1)$
    $cat_1 \leftarrow \text{Concat}(u_1, x_1)$
    $x_7 \leftarrow \text{dec1}(cat_1)$
    $\hat{y} \leftarrow \text{Sigmoid}(\text{Conv1x1}(x_7))$
    return $\hat{y}$
end procedure

The proposed 'Vanilla U-Net' model adopts a symmetric encoder-decoder architecture, tailored for image-to-image translation tasks such as synthetic bokeh generation. The forward pass begins with a sharp RGB input image tensor, which is progressively encoded through four convolutional blocks: 'enc1', 'enc2', 'enc3', 'enc4'. Every encoder block consists of two convolutional layers with batch normalization and ReLU activations, followed by a 2x2 Max pooling to reduce spatial resolution while increasing feature abstraction.

At the deepest level ('enc4'), the model is able to capture the most compact, semantically rich representation of the image. From there, we begin the decoding phase via transpose convolutions and skip connections. At each decoder level, we ensured that the upsampled feature map is concatenated with its corresponding encoder output to preserve fine-grained spatial detail, a key element that

is required to reconstruct high-frequency features like bokeh contours. Padding is then applied to ensure shape compatibility for concatenation.

Our final output is generated when passed through the 1x1 convolution followed by a sigmoid activation to ensure pixel values remain in the [0,1] range. This design will enable the model to synthesize a realistic bokeh while maintaining our structural image features.

4.3 Loss Function

The loss function used in our model is a simplified version inspired by that of PyNET's paper. The aim of our loss function is to guide the network to produce an output that is simultaneously close to the target bokeh image in terms of the pixel value, structure, and human perception. We utilise a weighted perceptual loss which leverages a Gaussian pyramid scale and applies different weights to the foreground (in-focus) and background (out-of-focus) regions of the image. This is to encourage subject sharpness and allow for more pixel deviation in the intended bokeh blur. The L_1 loss enforces pixel-level accuracy and colour fidelity. The L_{SSIM} loss focuses on preserving structural information between image patches. The L_{LPIPS} loss addresses perceptual realism through using deep features from a pre-trained AlexNet to measure similarity in a way that aligns better with human perception.

The key difference between our loss strategy and PyNET's is that PyNET uses sequential, level-by-level training and thus also level-dependent loss functions. In comparison, our approach uses a simplified, single combined loss for the end-to-end training of our Vanilla U-Net architecture.

$$\text{Total Loss} = \lambda_1 L_1 + \lambda_2 L_{SSIM} + \lambda_3 L_{LPIPS} \tag{1}$$

where default loss functions for all VBase models in Table 1 & 2 use: $\lambda_1 = 1$, $\lambda_2 = 1$, $\lambda_3 = 0.5$

The primary objective of our models is to seek to minimise this total loss function as the model learns and trains further. We treat the minimisation of validation loss as the most important evidence of learning and utilise this qualification in Section 5.2 when selecting optimal model architectures.

4.4 Learning Rate

As we learned in the ST311 course, learning rate is considered to be perhaps the most important hyperparameter - "if you have time to tune only one hyperparameter, tune the learning rate" [14].

In Section 5.2 we kept constant learning rates to try to find the best starting learning speed given our compute constraints. This identified 2e-4 (0.0002) as the optimal for our architecture, allowing significant learning to occur within just 10-30 epochs on a T4 GPU due to its aggressive value compared to typical CNN learning rates [1]. However, in our final training run in Section 5.3 we used a more advanced, dynamic learning rate. The final Vanilla U-Net model used a learning rate scheduler with a patience level of 3 epochs and a decrease of 20% in the learning rate occurring if validation loss did not decrease further in 3 epochs. This dynamic learning rate was used to surpass plateaus in learning and improve our model's performance, as can be seen in the description of Figure 2.

4.5 Addressing Underfitting and Overfitting

To address overfitting and underfitting in our models we followed the approach shared in week 4 of the ST311 lecture series [5]:

1. Implementing normalisation
2. Adding data augmentation
3. Tweaking learning rate
4. Increasing regularisation further with weight decay
5. Trying different SoTA model architectures
6. Increasing/decreasing hyperparameter model size through changing width and depth values.

5 Results

5.1 Observations and Limitations

5.1.1 Underfitting observations

One significant concern we noticed in the experiments presented in this section was that the validation loss remained consistently lower than training loss. This was found to be a puzzling insight into the difficulties of data distributions in complex, not-easily-identifiable distributions such as that of images. Originally we mistakenly performed our smaller mini training loops on the subset of our train data and used the testing data as our validation set. Although we later corrected this error and used the correct testing data and instead used a 20% random split of our training data for our validation set, we saw that our models were correctly overfitting on the test data when used as the validation set. However, after using the 20% validation split set we found that our models all underfitted, as seen in Table 1 and 2. Unfortunately due to the complexity in identifying both difficulty and diversity in images, we are not able to say with full certainty whether our models now underfit due to the validation set containing intrinsically easier examples to blur, or whether our models were just too simple for the task at hand.

As per the typical training and validation approach shared in the ST311 lectures we aimed initially to overfit our model and perform regularisation to reduce this overfitting [5]. However, as seen from our results in Table 1, when we tried to increase our model's complexity by increasing the depth and thus subsequent parameter count from a 0.47/1.86M level to 7.70M, our underfitting concern remained. One thing to note is that a longer runtime, as seen by VBase_3 with 30 epochs, did yield better convergence and reduced gap between the training loss and validation loss, but this may have been due to the model simply learning the training data over time and still being too simple to tackle a bokeh generation task. In comparison to PyNET's model, our VBase_3 model was trained on only a subset of the training and validation data, and has 16.2% of PyNET's total parameters. This was due to compute and time constraints, but appears as a likely cause of our model's results.

5.1.2 Addressing the Underfitting

Upon further research into whether our model was truly underfitting, we did note that our quantitative results remained competitive and even comparable to PyNET's, suggesting adequate learning capacity nevertheless. In addition, typical underfitting leads to a plateau, however our model appeared to converge as it ran for more epochs, as seen in Figure 2. This led us to explore additional checks on whether our model was truly underfitting, such as reducing regularisation effects which are applied to training data, but not validation data, thus potentially causing our puzzling results [8]. In Table 1's VBase_2 experiments it can be seen that when we removed batch normalisation the generalisation gap did decrease due to the fact that batch normalisation can create slight noise during training by using batch statistics instead of running statistics. While the gap reduced, both our train and validation losses increased significantly, indicating that BatchNorm provided crucial regularisation benefits. Another likely culprit is the complex multi-level loss function with focus masking that we used, which may have created more optimisation challenges on the training set.

Despite this underfitting phenomen, we proceeded with model training due to the strong performance metrics on the validation and test data nevertheless, given the overarching goal of minimising validation loss and yielding reasonable qualitative results with a model 16.2% the size of PyNET - seen in Figure 1 and 3.

5.1.3 Hardware and Software

The code for this project was implemented with the PyTorch framework. Our models were trained for 10-30 epochs on a single Nvidia Tesla T4 GPU on the free Google Colab plan. Unfortunately this did not allow for more rigorous exploration of training larger sized, more complex models, such as the 47.5M size of PyNET. In addition, we do not have direct knowledge of the total training time of the larger PyNET counterpart and only know they utilised a V100 GPU with 20-100 epochs used for hyperparameter optimisation [2]. Hence, we cannot say for certain whether the simplicity of our model was the direct cause of the identified underfitting.

5.2 Architecture and Hyperparameter Experimentation

In this section we explore different model sizes and hyperparameter combinations, to optimise our CNN's learning before launching the full, longer training run in Section 5.3.

Table 1: *This table showcases the various hyperparameter tuning that took place for the various models*

	Data_ Aug	B_norm	Width	Depth	Lr	B_s	Epochs	Params	Runtime	Train_ Loss	Val_Loss	Gap Magnitude
VBase_1	N	Y	64	3	2e-4	4	10	1.86	35	0.66	0.59	0.07
VBase_2	N	Y	32	3	2e-4	4	10	0.47	19	0.69	0.62	0.07
		N	32					0.47	18	1.00	0.97	0.03
			64	4	5e-4	8		7.70	43	0.65	0.60	0.04
VBase_2	N	Y	32	3	2e-4	4	10	0.47	19	0.69	0.62	0.07
					5e-4				17	0.80	0.71	0.08
					1e-4				18	0.82	0.73	0.08
							30		50	0.71	0.66	0.05
with wd									17	0.80	0.70	0.10
VBase_3	**N**	**Y**	**64**	**4**	**2e-4**	**4**	**30**	**7.70**	**134**	**0.58**	**0.59**	**0.01**

Table 1 is a log of some of the key optimisation tests we performed for this project, as per the iterative methodology mentioned in Section 4.5. Some key insights from our experimentation are that our higher parameter models improved model performance but were significantly more computationally expensive as shown by VBase_3's over 2 hour runtime on just a small subset of the train and validation data. In addition, a more aggressive learning rate of 2e-4 was found as a good balance between runtime and validation loss performance.

Table 2: *This table showcases the best performing model after all tests concluded in Table 1*

	Data_ Aug	B_norm	Width	Depth	Lr	B_s	Epochs	Params	Runtime	Train_ Loss	Val_Loss	Gap
VBase_1	N	Y	64	3	2e-4	4	10	1.86	35	0.66	0.59	0.07
VBase_2	N	Y	32	3	2e-4	4	10	0.47	**19**	0.69	0.62	0.07
VBase_3	N	Y	64	4	2e-4	4	30	7.70	134	0.58	**0.59**	**0.01**

As seen from the results in Table 2, our VBase models show improved learning as the number of Epochs and thus runtime, increase. At the time our VBase_2 model was highlighted as our best performing model at minimising validation loss given its very low runtime, and thus efficiency. However, after running the VBase_2 architecture for a longer 30 epoch length (as seen in Table 1) we have concluded VBase_3's performance is superior given its small generalisation gap and its lowest validation loss. Therefore, our final training run and full evaluation on the unseen test data in Section 5.3 used the VBase_3 architecture.

Table 3: *This table showcases the L1, PSNR, SSIM, LPIPS losses for our 'most efficient' model (VBase_2) over 10 epochs and the effects of changing the weights of the loss function. We selected VBase_2 for these experiments given its fast and efficient train time.*

	λ_1	λ_2	λ_3	Epochs	Runtime	Train_Loss	Val_ Loss	L1	PSNR	SSIM	LPIPS
VBase_2	1	1	0.5	10	19	0.69	0.62	0.15	23.09	0.81	0.21
LPIPS Weighted	1	1	1	10	19	0.91	0.81	0.15	23.01	0.81	0.21
LPIPS Heavy	1	1	2	10	19	1.14	1.02	0.15	23.00	0.81	0.20
SSIM Heavy	1	4	0.5	10	19	2.30	2.09	0.15	22.91	0.81	0.23

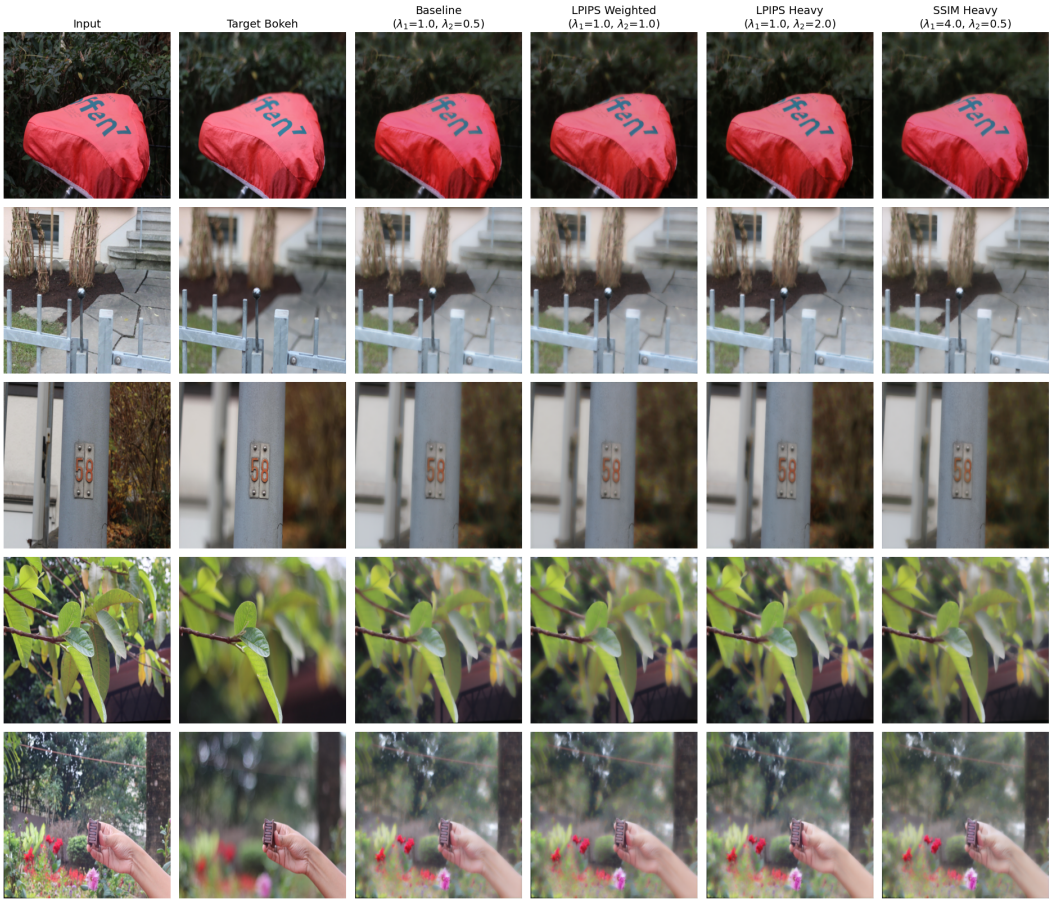

Figure 1: Qualitative results of varying loss weights on image outputs

Table 3 and Figure 1 showcase the quantitative and qualitative results of different loss weights on our VBase_2 model's performance, demonstrating the subtle bokeh rendering effects of varying loss weights. It must be noted that the performance of these different weighted models were not spectacular in qualitative performance and all exhibit some blurring of the foreground as well as the expected background. However this is plausible due to VBase_2's extremely small 0.47M parameter count and that it was trained for only 10 epochs for 19 minutes. Interestingly the LPIPS Weighted and Heavy configurations performed similarly to the Baseline, with only slight differences in the background intensity and colour fidelity. In contrast, the SSIM-Heavy configuration exhibits worse over-smoothing in textured regions (see rows 1 and 4), a loss of edge sharpness, and some inconsistent blur. Overall the baseline and LPIPS Weighted and Heavy configurations maintained better detail preservation while achieving more natural-looking bokeh.

5.3 Full Vanilla U-Net Model

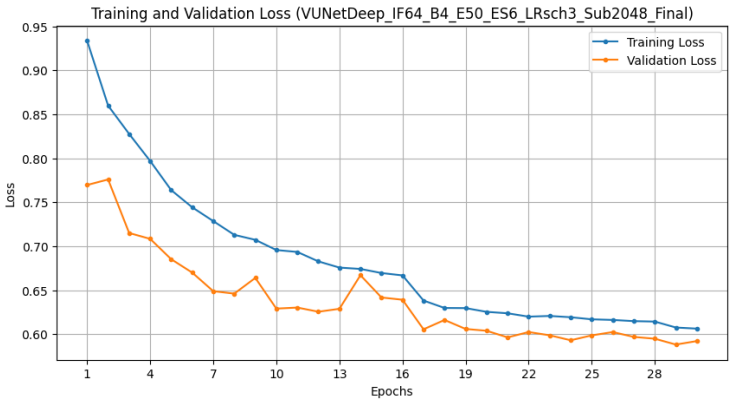

Figure 2: Our final Vanilla U-Net's training and validation loss over epochs on a larger data subset

Figure 2 showcases the consistent underfitting of our likely 'too simplistic' model over all 30 epochs, as discussed in Section 5.1. Here we can also see the dynamic learning rate working through training, as mentioned in Section 4.5. At epoch 15 the patience level 3 was reached, reducing the learning rate by 20% and thus helping yield the lowest minimum validation loss in epoch 17. This rate continued until epoch 27 when learning plateaued and the learning rate was further reduced, giving the final lowest validation loss in epoch 29. If we continued training further we would hopefully see continued strong performance through the leveraging of this dynamic learning rate, especially given most image tasks typically use significantly lower learning rates [1]. Having said this, from the curve it is clear to see that most learning happened at the earlier epochs given the flattening gradient as runtime continued.

Table 4: Table 4: *Comparison of the performance of our best model (VBase_3) versus PyNET on various metrics*

	Epochs	Params, M	Train runtime, m	PSNR ↑	SSIM ↑	LPIPS↓
Vanilla U-Net Full Run	30	7.70	250	**24.12**	0.8568	**0.1532**
PyNET	?	47.5	?	23.28	0.8780	0.2438

For our final Vanilla U-Net model seen in Table 4 we trained our model using the optimal VBase_3 configuration found in Table 1, but now for over 4 hours for 30 epochs. This runtime was $\sim$ 2x slower than our VBase_3 model from Table 1, given we increased the train dataset from 1024 to 2048 images, and increased the validation subset from 180 to the full 930 images for the final loop. We used this larger data subset for the final training for more stable training and better generalisation, whereas our mini loops used the smaller subset for quick hyperparameter experimentation.

Our quantitative results in Table 4 and qualitative results in Figure 3 confirm one of the observations discovered in PyNET's paper: "contrary to our expectations, neither of the numerical metrics work on the considered bokeh effect rendering problem" [2]. Although these metrics and their respective weighting in the loss function (see Figure 1) can have an effect on performance, the qualitative performance seems less related to PSNR, SSIM or LPIPS. Our final Vanilla U-net run in Table 4 shows superior performance in PSNR and LPIPS (bolded) compared to PyNET, reflecting the 'competitive' results of our model with significantly lower computational resources.

Figure 3: Qualitative comparison of our Vanilla U-net model versus PyNET

As seen in Figure 3, PyNET's outputs are far superior in qualitative quality compared to our Vanilla U-Net. PyNET's images maintain softer and better distributed bokeh blur across both image types. In addition, PyNET maintains a stronger ability to distinguish foreground (in-focus) vs background (out-of-focus) areas, with a sharper rendering of the foreground. Our Vanilla U-Net struggles more in this regard, with a slight blurring of the lemon and foreground leaves in the first image, and a softer effect on the railings in the second image.

Having said this, given our Vanilla U-Net's 6.2x fewer parameters and its low runtime of only 4 hours on a singular T4 GPU trained on just 44% of the dataset, we believe Figure 1 and 3 showcase our model's ability to successfully learn how to distinguish between the desired in-focus and out-of-focus regions.

6 Conclusion

In this work, we presented our Vanilla U-Net, a more efficient adaptation of the PyNET architecture, removing 3 of the 7 layers and yielding a 6.2x smaller model size for synthetic bokeh generation. For bokeh generation tasks, our model can be trained significantly faster than larger sized models. On the EBB! Dataset, we achieved superior results to PyNET in 2 out of 3 quantitative performance metrics. In addition, we performed qualitative analysis into how the weighting of various measures can affect final output images, and we showcased the superior performance of PyNET qualitatively, albeit with significantly more training time. We remain excited about the future of synthetic bokeh generation, particularly through its possible applications in improving consumer-grade photography hardware, such as that found on mobile phones.

Extending on from the limitations mentioned in Section 5.1, our underfitting concern remained present due to the lack of compute and time available to test a larger parameter model and compare the respective performance. Furthermore, we lost some time due to the incorrect naming of the EBB!'s testing data as the validation data. In addition, we found difficulties with using the PyNET model from their GitHub and identified other users had similar issues. Therefore, this made it difficult to provide more qualitative comparison results than those already provided. Future research could involve emulating the PyNET 7-level model in the PyTorch package with updated code to

allow for easier comparison. This comparison should also record how long each model requires to be trained, as this was an aspect missing from the original PyNET paper [2].

Future work should extend the size and complexity of the Vanilla U-Net further to try to find a better middle ground between our low model parameter count of 7.70M and PyNET's larger 47.5M parameter model. This additional research would likely provide a better insight into when the model starts from an overfitting stand point, and gradually learns and improves on the validation set. As a result, one would be able to find the best trade off inflection point between model size and complexity, compared to training time. Therefore, this would provide a stronger insight into whether the model's size, architecture, or task complexity were the bottlenecks in our underfitting performance. Alongside this, we recommend a better splitting of 'complex scenes' in the EBB! Dataset to allow for the potential exploration of more modern architectures such as Transformers and Attention mechanisms to address one of the concerns mentioned in the PyNET paper: "according to the obtained visual results, around 5-8% of the rendered bokeh images might contain strong visible artifacts. The problems usually happen in complex scenes with multiple objects that have either similar colors or are occluded by each other" [2]. We attempted exploring these Attention-based architectures and found better performance from our original Vanilla U-Net in 10 epoch training runs. However, due to time and compute limitations we could not provide extensive research into this area and recommend this for further study.

Overall, there are various potential exploration areas for improving synthetic bokeh, with exciting applications potentially improving the software processing of camera applications in the largest photography market there is, smartphones.

References

[1] Gong, Y., Zhan, Z., Jin, Q., Li, Y., Idelbayev, Y., Liu, X., Zharkov, A., Aberman, K., Tulyakov, S., Wang, Y. & Ren, J. (2024). *E2GAN: Efficient Training of Efficient GANs for Image-to-Image Translation*. arXiv preprint arXiv:2401.06127v1.

[2] Ignatov, A., Patel, J. & Timofte, R. (2020). *Rendering Natural Camera Bokeh Effect with Deep Learning*. arXiv preprint arXiv:2006.05698.

[3] Shorten, C. & Khoshgoftaar, T.M. (2019). *A survey on Image Data Augmentation for Deep Learning*. Journal of Big Data, 6(1), p. 60.

[4] Khan, A., Sohail, A., Zahoora, U. & Qureshi, A.S. (2019). *A Survey of the Recent Architectures of Deep Convolutional Neural Networks*. arXiv preprint arXiv:1901.06032.

[5] ST311 Week 5: Black Belt Training [Class slides], ST311 XAI: Advanced Machine Learning, 11 February 2025.

[6] Isong, N. (2025). *Building Efficient Lightweight CNN Models*. arXiv preprint arXiv:2501.15547v1.

[7] Güven, S.A., Şahin, E. & Talu, M.F. (2024). *Image-to-Image Translation with CNN Based Perceptual Similarity Metrics*. Big Data and Business Intelligence, 9(1), pp. 84–98.

[8] Abusaleh, S. (2024). *Why My Training Loss is Higher Than Validation Loss? Is the Reported Loss Even Accurate?* Medium, 25 June [online].

[9] Azad, R. et al. (2022). *Medical Image Segmentation Review: The Success of U-Net*. arXiv preprint arXiv:2211.14830.

[10] Luo, X., Peng, J., Xian, K., Wu, Z. & Cao, Z. (2023). *Defocus to Focus: Photo-realistic Bokeh Rendering by Fusing Defocus and Radiance Priors*. arXiv preprint, arXiv:2306.04506.

[11] Peng, J., Pan, Z., Liu, C., Luo, X., Sun, H., Shen, L., Xian, K. & Cao, Z. (2023). *Selective Bokeh Effect Transformation*. in NTIRE 2023 Workshop on Visual Effects at CVPR 2023, pp. 1456–1465.

[12] Lee, B., Lei, F., Chen, H. & Baudron, A. (2022). *Bokeh-Loss GAN: Multi-stage Adversarial Training for Realistic Edge-aware Bokeh*. arXiv preprint, arXiv:2208.12343.

[13] Fortes, A., Wei, T., Zhou, S. & Pan, X. (2025). *Bokeh Diffusion: Defocus Blur Control in Text-to-Image Diffusion Models*. arXiv preprint, arXiv:2503.08434v1.

[14] Brownlee, J. (2020). *Understand the Dynamics of Learning Rate on Deep Learning Neural Networks*. Machine Learning Mastery, 12 September.

A Acronym definitions

For our tests performed in Table 1, 2, and 3, the below acronyms were used for the column titles and have their respective definitions listed in more detail below:

With wd = indicates weight decay

Data_Aug = data augmentation, transforming, flipping, etc. (Y for yes, N for no)

B_norm = whether batch normalisation was added

Width = Width, Initial Features (init_features) controlling network width

Depth = Depth, number of encoder/pooling levels

Lr = learning rate

B_s = batch size

Epochs = Epochs

Params = total parameters in Millions

Runtime = runtime in minutes total

Val_Loss = final validation loss (is a weighted sum of L1 loss, PSNR, SSIM, and LPIPS)

Gap = magnitude of |val_loss - train_loss|