Supplementary Materials
Investigating Neural Architectures by Synthetic Dataset Design

Adrien Courtois[*], Jean-Michel Morel, Pablo Arias
Centre Borelli, ENS Paris-Saclay
4 Av. des Sciences, 91190 Gif-sur-Yvette, France
adrien.courtois@ens-paris-saclay.fr

1. Code and datasets

The code and the datasets are available to download at this URL.

2. Description of the layers

2.1. Global Context Layer

This layer has been designed based on thorough experiments on the Non-Local Networks introduced by Wang [13], a network using a non-local operation resembling self-attention [11]. They observe that the non-local operation of [13] suffers a kind of mode-collapse where most of the attention maps are equal. The Global Context Layer aims at directly computing this shared attention map and alleviating at the same time the quadratic cost.

We changed the original implementation by processing the output by an inverted bottleneck and by using GELU non-linearities and Layer Scale within each skip-connection. We also extended the layer by introducing a mechanism of head within the layer, but we didn't use it in this paper. The original layer reduces the number of channels of the input feature map before processing it. We kept this mechanism and used a reduction factor of $r = 2$ as suggested in the paper.

2.2. Global Average Pooling

To further highlight the need for non-locality, we chose to assess the performance of the simplest non-local layer. The most straight-forward way to aggregate cues from the entire image is to compute the mean using a Global Average Pooling layer *i.e.* a spatial mean of each channel independently. This spatial mean is added to each spatial location of the input feature map and is processed by an inverted bottleneck with GELU non-linearities. Layer Scale and a pre-normalization using a LayerNorm is also used.

[*]This work was supported by grants from Région Ile-de-France.

2.3. Deformable Convolutions

This candidate non-local layer has been intensively used in the literature [5, 12]. It allows to break the usual rectangular shape of the convolution kernel in a learnable fashion. It results in sparse, continuous convolution kernels. Due to the computational cost of this layer, we only used it one in every scale of the encoder and the decoder. We used the implementation of TORCHVISION. As for every nonlocal layer, we processed the output with an inverted bottleneck, we used skip connections with Layer Scale and GELU non-linearities. The masks and offsets were computed using the ConvNeXt block developed in [6].

2.4. Lambda Layer

The Lambda Layer was introduced for image classification as a linear approximation of self-attention [11]. In its original formulation, it is a local operation, applied on 23×23 patches of the input feature map. Part of this limitation comes from a learned positional embedding whose size scales quadratically with the spatial dimensions.

To apply this layer on the entire image, we replaced the learned positional encoding by a non-learnable cosine embedding [11]. The input to the layer is pre-normalized before passed through the layer and the output is processed by an inverted bottleneck with GELU non-linearities. We also used skip connections with Layer Scale. Furthermore, we fixed the amplitude problem highlighted in the original paper by dividing the output by $\sqrt{C'}$, where C' is the dimension of K and Q. We set $h = 4$ and $C' = 16$ as suggested in the original paper.

This fix can be justified theoretically: in the Lambda layer, we compute a scalar product between the lines of Vx and the columns of Qx. At initialization, Q and V are initialized such that if $x \sim \mathcal{N}(0,1)$, then $Vx, Qx \sim \mathcal{N}(0,1)$. Now, if we consider $\boldsymbol{X} = (X_1, \ldots, X_n)$, $\boldsymbol{Y} = (Y_1, \ldots, Y_n)$ such that $X_i, Y_i \sim \mathcal{N}(0,1)$, then $\mathbb{E}[\langle \boldsymbol{X}, \boldsymbol{Y} \rangle] = 0$ and $\mathbb{V}\mathrm{ar}[\langle \boldsymbol{X}, \boldsymbol{Y} \rangle] = n$. This high variance introduces high amplitudes within the feature maps, which makes for instability. Therefore, dividing the scalar product by $\sqrt{n}$ reduces the variance of the output and fixes the instability.

2.5. LambdaTT

This layer is built on top of the Lambda layer described in Section 2.4. First, let us recall the computations made within the Lambda layer. Given three matrices $(Q, K, V) \in \mathbb{R}^{M \times C_{\text{in}}} \times \mathbb{R}^{M \times C_{\text{in}}} \times \mathbb{R}^{C_{\text{out}} \times C_{\text{in}}}$ and an input feature map $x \in \mathbb{R}^{C_{\text{in}} \times N}$, the lambda layer first computes an attention map:

$$\bar{K} = \text{SOFTMAX}_N(Kx).$$

This attention map guides the computation of the subsequent features and is crucial for the generalization capabilities of the layer. However, this attention map is computed based on the similarity between the lines of K and the columns of x. The matrix K being learned, the attention map is necessarily guided by the statistics it learned during the training process and because of that, the attention map is not input-dependent enough. In the other hand, the idea of the Lambda layer is to compute a matrix λ_{content} to be applied to each spatial location of the input individually:

$$\lambda_{\text{content}} = \bar{K}(Vx)^T,$$

$$y_{\text{content}} = \lambda_{\text{content}}^T Qx.$$

This matrix λ_{content} is largely input-dependent and therefore, we propose to replace K by such matrix λ:

$$\bar{K}_1 = \text{SOFTMAX}_N(K_1 x),$$

$$\lambda_{\text{mask}} = \bar{K}_1(V_1 x)^T,$$

$$\bar{K}_2 = \text{SOFTMAX}_N(\lambda_{\text{mask}} K_2 x),$$

$$\lambda_{\text{content}} = \bar{K}_2(V_2 x)^T,$$

$$y_{\text{content}} = (\lambda_{\text{content}})^T Qx.$$

These computations define the LambdaTT layer. It amounts to introducing two additional matrices $K_1 \in \mathbb{R}^{M \times C_{\text{in}}}$ and $V_1 \in \mathbb{R}^{M \times C_{\text{in}}}$ and two additional computations which scale linearly with the input's dimensions.

This implementation can be seen as an iteration of the K-mean algorithm: the barycenters are computed once within λ_{mask}, the mapping of each datum to its closest barycenter is done within $\bar{K}_2$ and λ_{content} contains the updated barycenters.

3. Training procedures

3.1. Optimizer

We trained all of our networks with a custom optimizer built on top of Ranger21 [14]. It is a composition of Positive-Negative momentum [15] with $\beta_2 = 1$, AdaBelief [18], decoupled weight decay [7], LookAhead [17] with a merge time of 5 and $\alpha = 0.5$, gradient centralization [16], adaptive gradient clipping [2] with the original hyper-parameters and a softplus calibration [10] with $\beta = 50$. We used the same value of $5 \cdot 10^{-4}$ for the weight decay parameter for all of the networks.

On top of these tweaks, we found that further dividing the gradient of each parameter by its norm (therefore doing a normalized gradient descent [3]) helps improving the results by a very large margin. In practice, we found that using this simple trick alongside with the structural modifications of [6] yielded a $10\times$ improvement on the test loss when compared with the original U-Net trained with AdamW when trained for the same number of epochs.

3.2. RDE dataset

All of our networks were trained for 50 epochs. We used a linear warmup [8] following the recommendations of the original paper and a linear warmdown for the last 14 epochs. The training dataset contains 50,000 images for training and 12,500 images for testing. No data augmentation was used. To ensure reproducibility and simplicity we used **no further trick**.

For each network, we tested up to five different learning rates and kept the one yielding the best results.

Although multiple losses have been proposed in the literature, we trained all our networks with the Scale-Invariant loss [1, 4] that we chose for its simplicity.

As the width of our networks is shared across all scales, we chose the width of each Non-Local U-Net so that its number of parameters is as close as possible to the baseline U-Net. We did not evaluate any other commonly-used architecture for depth estimation. As they all use a backbone, the objective of parameters was impossible to reach. All the experiments were made using a single Tesla V100 32GB GPU and a batch size of 32.

3.3. Centered Square

For this dataset, we encountered multiple optimization-related problems. This is due to the fact the dataset only contains 484 training images. We managed to obtain stable trainings using the Ranger21 [14] optimizer with the same linear warmup and linear warmdown as for the RDE dataset, and we trained for 100 epochs. In particular, the Ranger21 optimizer features no weight decay and a variant of the normalized gradient descent. The loss we used was the MSE and the batch size was 32. We tuned the learning rate for each configuration but found that $5 \cdot 10^{-3}$ consistently yielded the best results.

4. ColorCode

This dataset contains 20,000 training images and 10,000 testing images and therefore, we didn't encounter any

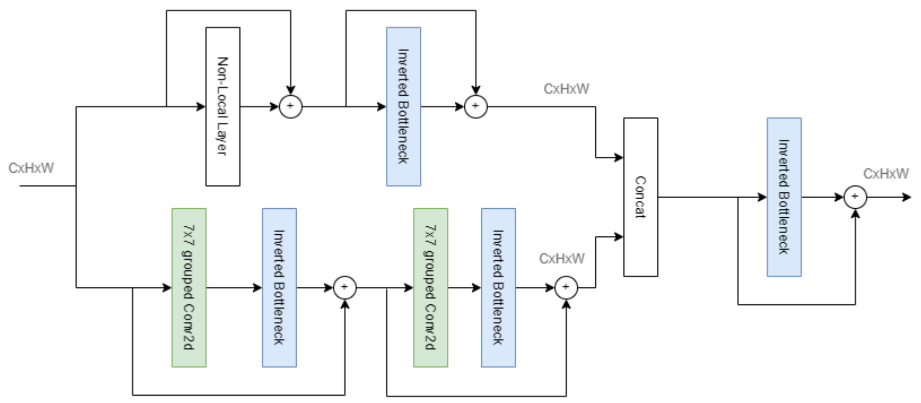

Figure 1: Structure of the layer used at each scale of the Non-Local U-Net, replacing the double convolution of the original U-Net. This enables the incorporation at each scale of the U-Net of any non-local layer. The local and non-local feature computations are done in parallel.

optimization-related issue. We trained all the networks with the optimizer described in Section 3.1. The loss to minimize was the MSE. We used a batch size of 256. We used the same learning rate scheduling as for the RDE dataset.

To tune the hyper-parameters, we firstly trained the networks with learning rates in $\{10^{-1}, 5 \cdot 10^{-2}, 10^{-2}, 5 \cdot 10^{-3}, 10^{-3}\}$. Then, we tested four other learning rates in-between the learning rates yielding the two best results. Most of the optimal learning rates found this way were in the $[5 \cdot 10^{-3}, 10^{-2}]$ range.

All of the networks used eight heads and a width of 256 and had roughly the same number of parameters.

5. Algorithm for generating RDE

The RDE synthesis algorithm is split into three steps. First, the rectangles are generated by a set of rules that carefully eliminate ambiguous cases. Then, all pairs of potentially intersecting rectangles are compared to each other to infer if one of them overlaps the other. Finally a *depth-first search* algorithm derives the global scene organization and fixes the unambiguous ground truth, each rectangle receiving its lowest possible integer rank compatible with all pairwise overlap observations.

The first algorithm consists in a WHILE loop, randomly generating coordinates for a candidate rectangle and checking for the following constraints:

- The candidate should not be too small *i.e.* have at least a width and height of $W/10$ and $H/10$ respectively, where (H, W) is the dimension of the image to be generated.

- The candidate should not occlude another previously-generated rectangle too much. This translates to constraining the maximum number of pixels belonging to a given rectangle in each row and column of the image to be larger than n_{vis}. We also check for the minimum number of visible pixels in each row and column and impose it to be larger than n_{gap}.

- To avoid the near-ambiguous case where translating a rectangle by one pixel would change the ground truth, we check that the candidate does not share a side with another already-generated rectangle. We also prevent the case where translating the candidate by one pixel would result in a side being shared.

- We constrain two parallel sides belonging to two different rectangles to be separated by at least n_{par} pixels.

- Finally, a T-junction between two rectangles must be separated by at least n_{vis} pixels from the closest edge.

All of these constraints ensure the *genericity* of the scene, namely that a small perturbation of the rectangles positions would not alter scene interpretation. If all the conditions are met, the candidate is added to the list of rectangles. The loop stops when $n_{\text{rectangles}}$ (usually 10) have been generated or after 1000 unsuccessful trials (which happens about once every 500 runs).

Once the $n_{\text{rectangles}}$ are generated, we perform a pairwise comparison of the rectangles. Consider a pair of rectangles (A, B) where B has been generated before A. If an edge of A forms the leg of a T-junction with an edge of B and if this T-junction is not occluded by another successive rectangle, then B is set to be above A. Then, for each rectangle of the pair we compute the smallest rectangle that contains all the visible pixels belonging to it. Let's call these two rectangles C and D. If C and D overlap and if there is a pixel belonging to A in the intersection *i.e.* if the intersection is not occluded, A is deduced to be above B.

Then, we consider the one-to-one comparisons as a graph and do a *depth-first search* to determine the global ordering of the square.

Finally, we randomly attribute to each rectangle a color from a fixed list of $n_{\text{rectangles}}$ distinct colors generated once and for all by uniformly sampling the hue space. The generated image is blurred by an antialiasing Gaussian filter with standard deviation 0.7. The ground truth is normalized to have values in $[0, 1]$, as per common practice.

Despite the different optimizations, the resulting algorithm is quite slow: approximately eight CPU hours are required to generate 62,500 image-ground-truth pairs. We further parallelize this generation using multi-threading on sixteen CPU cores. The final algorithm takes mere minutes to generate the entire dataset.

We set $H = W = 128$, $n_{\text{vis}} = n_{\text{gap}} = n_{\text{par}} = 5$. Decreasing any of the aforementioned parameters amounts to making the task harder. We could also have chosen to sample a different set of colors for each samples. This way the network would have the additional difficulty to learn the colors from each sample.

6. Bonus: getting your positional encoding to the next level

Potential sources of improvement The positional encoding described in the paper is, given $K \in \mathbb{R}^{M\times N}$, $Q \in \mathbb{R}^{M\times N}$, $V \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}}$:

$$\bar{K} = \text{SOFTMAX}_N(Kx) \in \mathbb{R}^{M\times N},$$

$$y_{\text{pos}} = \bar{K}P^T P \in \mathbb{R}^{M\times N},$$

where $P \in \mathbb{R}^{C\times N}$ is a given positional encoding matrix. The matrix P is of the form:

$$P_{c,n} = \begin{cases} \cos(w_k n) & \text{if } c = 2k \\ \sin(w_k n) & \text{if } c = 2k+1 \end{cases}.$$

A first source of improvement can come form a better design of the family $(w_c)_{c\in[\![1,C/2]\!]}$ as described in the paper. One can further note that the value C could be anything. For instance, we found that in some cases using $C = 1024$ improved the performance. We have not used this value in the paper because it would lead to an unfair comparison between the different approaches.

Another source of improvement could come from using two positional encoding matrices $P_1, P_2 \in \mathbb{R}^{C\times N}$ such that:

$$y_{\text{pos}} = \bar{K}P_1^T P_2.$$

For instance, defining

$$P_1 = \begin{cases} \cos(w_k n) & \text{if } c = 2k \\ \sin(w_k n) & \text{if } c = 2k+1 \end{cases},$$

$$P_2 = \begin{cases} -\sin(w_k n) & \text{if } c = 2k \\ \cos(w_k n) & \text{if } c = 2k+1 \end{cases},$$

yields a non-isotropic encoding of the relative position of each pixel. This is not useful for the RDE dataset since the task is isotropic but could be of help for other tasks. This approach reassembles [9] but doesn't introduce any additional parameter nor computational cost.

How to use heads In order to incorporate heads within the layer, one has to reshape the matrix P into the size $(h, C/h, N)$, the tensor $\bar{K}$ into $(h, M/h, N)$ and compute

$$(y_{\text{pos}})_{h,k,n} = \bar{K}_{h,k,m}P_{h,c,m}P_{h,c,n},$$

using Einstein notations.

Two dimensional position encoding Finally, we would like to sketch the design of a 2D positional encoding following the same requirements. Indeed, multiple online implementations simply do not take into account the heads that could be used alongside the layer. Given a family $(w_c)_{c\in[\![1,C/4]\!]}$, the positional encoding is given by

$$P_{c,h,w} = \begin{cases} \cos(w_k h) & \text{if } c = 4k \\ \sin(w_k h) & \text{if } c = 4k+1 \\ \cos(w_k w) & \text{if } c = 4k+2 \\ \sin(w_k w) & \text{if } c = 4k+3 \end{cases}.$$

When used in conjunction with heads, one would ideally like to have C/h to be a multiple of 4.

References

[1] Shariq Farooq Bhat, Ibraheem Alhashim, and Peter Wonka. Adabins: Depth estimation using adaptive bins. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4009–4018, 2021. 2

[2] Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In *International Conference on Machine Learning*, pages 1059–1071. PMLR, 2021. 2

[3] Jorge Cortés. Finite-time convergent gradient flows with applications to network consensus. *Automatica*, 42(11):1993–2000, 2006. 2

[4] Jin Han Lee, Myung-Kyu Han, Dong Wook Ko, and Il Hong Suh. From big to small: Multi-scale local planar guidance for monocular depth estimation. *arXiv preprint arXiv:1907.10326*, 2019. 2

[5] Yanghao Li, Yuntao Chen, Naiyan Wang, and Zhaoxiang Zhang. Scale-aware trident networks for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6054–6063, 2019. 1

[6] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. *arXiv preprint arXiv:2201.03545*, 2022. 1, 2

[7] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 2

[8] Jerry Ma and Denis Yarats. On the adequacy of untuned warmup for adaptive optimization. *arXiv preprint arXiv:1910.04209*, 7, 2019. 2

[9] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021. 4

[10] Qianqian Tong, Guannan Liang, and Jinbo Bi. Calibrating the adaptive learning rate to improve convergence of adam. *Neurocomputing*, 2022. 2

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 1

[12] Xintao Wang, Kelvin CK Chan, Ke Yu, Chao Dong, and Chen Change Loy. Edvr: Video restoration with enhanced deformable convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019. 1

[13] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7794–7803, 2018. 1

[14] Less Wright and Nestor Demeure. Ranger21: a synergistic deep learning optimizer. *arXiv preprint arXiv:2106.13731*, 2021. 2

[15] Zeke Xie, Li Yuan, Zhanxing Zhu, and Masashi Sugiyama. Positive-negative momentum: Manipulating stochastic gradient noise to improve generalization. In *International Conference on Machine Learning*, pages 11448–11458. PMLR, 2021. 2

[16] Hongwei Yong, Jianqiang Huang, Xiansheng Hua, and Lei Zhang. Gradient centralization: A new optimization technique for deep neural networks. In *European Conference on Computer Vision*, pages 635–652. Springer, 2020. 2

[17] Michael Zhang, James Lucas, Jimmy Ba, and Geoffrey E Hinton. Lookahead optimizer: k steps forward, 1 step back. *Advances in Neural Information Processing Systems*, 32, 2019. 2

[18] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar C Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *Advances in neural information processing systems*, 33:18795–18806, 2020. 2