Report on using convolutional neural networks to estimate normalization coefficients for diffusion-based correlation operators

Guillaume Coulaud

September 21, 2022

Contents

Abstract

A correct modeling of the background error covariance matrix can have a very positive impact on oceanographic numerical modeling. When using diffusion-based covariance operators, computing normalization coefficients is crucial to ensure that the diagonal elements of the modeled correlation matrix are all equal to one. However, this is a computational bottleneck of the process. In this paper, we study a deep learning approach with convolutional neural networks (CNN) to estimate the normalization coefficients in a bi-dimensional setting. Our approach is validated on the coefficients from the Nemovar grid computed by a brute-force exact method. We achieved good results with a maximum absolute relative error of 4.3% and a mean of 0.4% averaged over 190 samples. On a simplified Python framework, the CNN approach gives error metrics twice as low as the classically used randomization method.

1 Introduction

The aim of this study is to provide a proof of concept of the use of deep learning methods for the estimation of the normalization coefficients of diffusion operators in the variational data assimilation. Current methods [1] for computing those coefficients are computationally costly, preventing the computation of the coefficient in each data assimilation cycle. The advantage of using neural networks (NN) to make inference is its low computational cost. However, the computation of the network weights during the training phase can be very expensive. This low computation cost makes NN an appealing alternative to existing methods, allowing to take into account the temporal variations of the model coefficients with seasons and ocean currents. As shown in [2, 3, 4], we use the universality of a convolutional neural network (CNN) to approximate a continuous function equivariant by translation with any accuracy, as long as the network is deep enough.

Let f be the function we want to approximate.

$$\begin{aligned} f\colon \mathbb{R}^{h\times w} \times \mathbb{R}^{h\times w} \times \mathbb{R}^{h\times w} &\to \mathbb{R}^{h\times w} \\ (A_i,\ A_j,\ W) &\mapsto \Gamma \end{aligned}$$

with A_i (resp. A_j) the scaled diffusivities on axis y (resp. x), W the volume of the cells, Γ the normalization coefficients, $h \times w$ the dimension of the grid.

The input of the neural network is $(A_i,\ A_j,\ W)$ which can be seen as a three-channel image.

By construction, the layers of a CNN are equivariant to translation [2]. This means that if the input of a function is translated, the output is translated in the same way, let f be a function and τ be a translation, we have $\tau \circ f(x) = f \circ \tau(x)$. For the function to approximate the equivariance to translation is obtained by using the scaled diffusivity and the volume of the cells as described in [5].

2 Problem definition

The first goal is to determine whether a CNN can reach the performance of a U-Net without exceeding the number of trainable parameters. To do that, we have a pre-trained U-Net with one million parameters and a CNN on which to perform hyperparameter tuning. The hypothesis to test is that the better performance of the U-Net can be explained by the larger number of parameters alone. Moreover, the problem is very local, which probably makes the use of a U-Net superfluous.

Subsequently, the goal is to beat the randomization method [1] on the Nemovar grid using a neural network.

To evaluate the performance, we compute the following metrics. Let us first define Γ the normalization coefficient, it is the target of the CNN, $\hat{\Gamma}$ the normalization coefficients estimated by the CNN, γ the vectorization of Γ noted $\gamma = \text{vec}(\Gamma)$, with vec the concatenation of the columns of the matrix, and $\mathcal{O}$ the set of indices i such that γ_i corresponds to the coefficient of a grid point in the ocean.

- For one sample $\Gamma \in \mathbb{R}^{h \times w}$,
  1. The relative error $\varepsilon \in \mathbb{R}^{|\mathcal{O}|}$ is defined component-wise as
  $$\varepsilon_i = \frac{\hat{\gamma}_i^2 - \gamma_i^2}{\gamma_i^2}, \ \forall i \in \mathcal{O}$$
  2. RMSE(ε): it gives the average accuracy of the model, it is defined by $\text{RMSE}(\varepsilon) = \sqrt{\mathbb{E}[\varepsilon \circ \varepsilon]}$ with $\circ$ the Hadamard product.
  3. The mean absolute relative error: $\varepsilon_{\text{mean}} = \dfrac{1}{|\mathcal{O}|} \sum_{i \in \mathcal{O}} |\varepsilon_i|$
  4. The quantile 99.99 of $|\varepsilon|$: ε_{Q} it gives the most inaccurate pixels
  5. The maximum absolute relative error: $\varepsilon_{\text{max}} = \max_{i \in \mathcal{O}} |\varepsilon_i|$
- For d samples $\Gamma^{(d)} : \mathbb{R}^{d \times h \times w}$
  1. The relative error on a sample ℓ: $\varepsilon^{(\ell)}$ such as $\varepsilon_i^{(\ell)} = \dfrac{\left(\hat{\gamma}_i^{(\ell)}\right)^2 - \left(\gamma_i^{(\ell)}\right)^2}{\left(\gamma_i^{(\ell)}\right)^2}, \forall \ i \in \mathcal{O}$,

     with $\gamma^{(\ell)} = \text{vec}(\Gamma^{(\ell)})$
  2. $\text{RMSE}(\varepsilon) = \dfrac{1}{d} \sum_{\ell=1}^{d} \sqrt{\mathbb{E}[\varepsilon^\ell \circ \varepsilon^\ell]}$
  3. The mean absolute relative error: $\varepsilon_{\text{mean}} = \dfrac{1}{d} \sum_{\ell=1}^{d} \dfrac{1}{|\mathcal{O}|} \sum_{i \in \mathcal{O}} |\varepsilon_i^{(\ell)}|$

4. $\varepsilon_{\mathrm{Q}} = \frac{1}{d}\sum_{\ell=1}^{d} Q(|\varepsilon^{\ell}|, 0.9999)$

5. The mean maximum relative error: $\varepsilon_{\max} = \frac{1}{d}\sum_{\ell=1}^{d} \max_{i\ \in \mathcal{O}} |\varepsilon_i^{(\ell)}|$

We consider three data sets generated using the brute-force method presented in [1], from artificial input fields (Ai,Aj,W). The first one comes from a python toy model (Mercator projection) and the other two from Nemovar grid (Partial_std, Full_std). Most of the work has been done on the Nemovar grid.

In practice, each data set is divided into three sets for training, validation, and tests. For the Python grid, we use 90 samples for training, 10 samples for validation, and none for the test. With the Nemovar data sets, there are 10 samples for training, 1 for validation, and 190 for the test. We used little data in training and validation for performance issues, as we wanted to reach 50,000 epochs to see if convergence was reached. In addition, it is interesting to see if the method works using few data because if the research is to be extended to the three-dimensional case, there will be fewer data than in 2D.

Both the Python and Nemovar grid are periodic in the East-West direction. However, for the Python grid, there is no North-South periodicity because there are two continents on the north and south of the grid. On the other hand, the Nemovar fields can be extended to the south and north of the domain, using a pseudo-periodicity property of the Nemovar grid; more details on the padding can be found in Folkes' report.

For the Nemovar grid, the difference between the two data sets is based on how we standardized the data. For the first one, Partial_std, we only standardized the image using the coefficients in the ocean, while for the second one, Full_std, we standardized the whole image. The standardization of Partial_std was a design error. This error was spotted around Section 5, and corrected afterwards by using Full_std. The change of normalization does improve the validation metrics. The difference between the two data sets is enlightened in Figure 1. The color map is saturated but the important thing to notice is that for the partial standardization the value for the continental cells is 0 while for the oceanic cells the values are both positive and negative. However, for the full standardization, the value of the majority of oceanic pixels is positive. We use the first data set until Section 5.

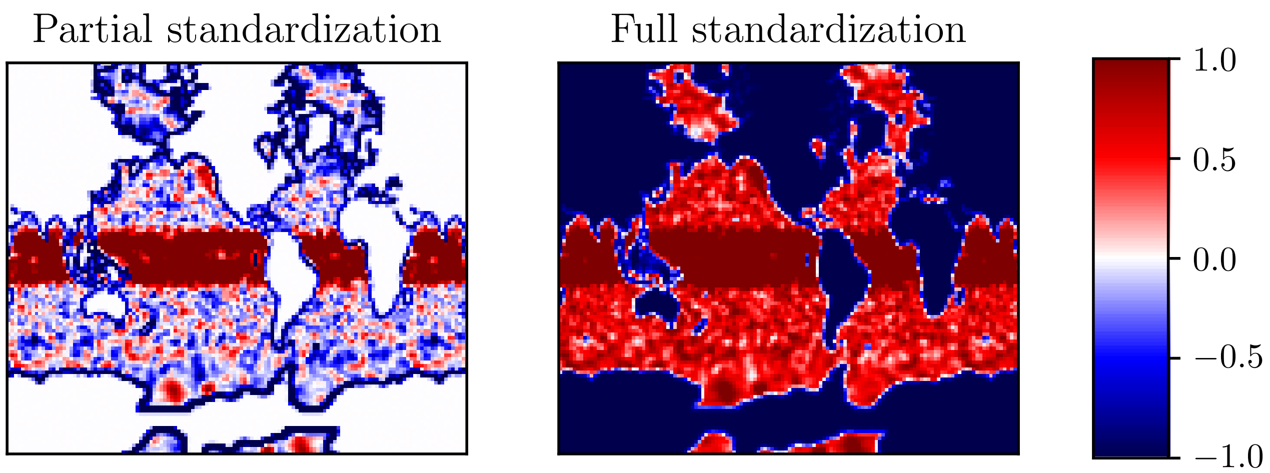

Figure 1: Comparison of the two data sets for the first channel, A_i, of the input data

To evaluate the performance of the CNN on the nemovar grid, we compare the metrics and the localization of the relative error for the test dataset. The test data set consists of all samples that are not used in any training, in this case, there are 189 samples.

For the Nemovar data sets, each training is done on the same number of training step with number of training step = number of epochs × number of samples ÷ batch size, we fixed that number at 5×10^6. This is useful as we will not have the same data set length when using data augmentation.

3 Choosing the architecture: CNN vs U-Net

The goal is to determine if the CNN can beat the U-Net using the Python grid. We tried a CNN with several layers and 3×3 convolution, batch normalization, Rectified Linear Unit ($\mathrm{ReLU}(x) := \max(0, x)$) as an activation function for the layers and $\mathrm{MSE}(\gamma)$ as a loss function. The different performances can be seen in Appendix A on Table 9, each metric is computed at the end of the 3000th epoch on the validation set. The optimal number of layers is found to be 10. This architecture has 3.36×10^5 trainable parameters.

Architecture	ε_{Q}	$\varepsilon_{\max}$	$\varepsilon_{\mathrm{mean}}$
U-Net	7.8×10^{-2}	1.3×10^{-1}	4.3×10^{-3}
CNN	2.3×10^{-2}	4.8×10^{-2}	9.5×10^{-4}

Table 1: Performance of CNN and U-Net on Python grid

The architecture we retain is a CNN with 10 layers composed of:

- 1 input layer:

  - 3×3 convolution
  - 3 input channels, 64 output channels
  - Batch normalization
  - ReLU

- 8 hidden layers:
  - 3×3 convolution
  - 64 input channels, 64 output channels
  - Batch normalization
  - ReLU

- 1 output layer
  - 3×3 convolution
  - 64 input channels, 1 output channels

Padding: in PyTorch the 2D convolution needs two parameters for the padding: *padding* and *padding mode*. We used a padding *same*, meaning that the size of the image is conserved through the forward propagation, the *padding mode* is *replicate* meaning the input tensor is filled using the replication of the input limit. Later on the Nemovar grid, we use padding *valid*, as we apply a 3×3 kernel for the convolution, the image size decreases by 1 for each layer.

Batch normalization: as defined in PyTorch documentation[1] it consists in normalizing the data along the channels then multiplying the data by one vector and adding another one, both vectors being learnable parameters of the CNN. We compared the performance of this architecture with the data set Partial_std, with and without batch normalization. The best results are obtained with batch normalization. For the rest of this report, we will always apply batch normalization in the input and hidden layers. It is also important to consider how the data were initially standardized. For example, when using Partial_std as the normalization uses the continental cells that were not used for the standardization, it might induce a loss of information. This could explain the difference in performance between the two data sets in Section 5

As we can see in Table 1, the CNN is better than the U-Net for all the metrics. The use of a CNN also considerably reduces the number of trainable parameters, 3.36×10^5 vs 1.6×10^7. As the results are promising and already better than the randomization with 10,000 samples, we will try to obtain similar results on the Nemovar grid. In order to achieve this, we stay on the most promising general architecture with the CNN and will try to improve it through data augmentation and by adding skip connections.

[1] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html

4 Finding a good baseline

In this part, we try to find a good configuration of hyperparameters before modifying the data with data augmentation or changing the architecture with skip connections. The various experiments are carried out on the date of the Nemovar grid.

4.1 Signed distance map to coast

As the errors are mostly located near the coast, we try to add in the input a signed distance map to coast to provide more information to the CNN. The signed distance is defined as the distance to coast for the ocean cells and the opposite of the distance for the continental cells. We compare the performance using the Manhattan distance, the Euclidean distance, and no signed distance input. As shown in Table 2 the best result is achieved with the Manhattan distance. We mainly look at ε_{Q} and ε_{max} as $\varepsilon_{\text{mean}}$ is already good, even if it is worse with the distance map. We trained a CNN on the data set Partial_std using 10 layers as described in Table 11, Experiment "Distance map". For the rest of the report, the Manhattan signed distance is used as input of the CNN. As we switched to the Nemovar grid, which is more complex than the Python grid, the scores are poorer than in Table 1.

	ε_{Q}	ε_{max}	$\varepsilon_{\text{mean}}$
No map	7.29×10^{-2}	1.18×10^{-1}	4.33×10^{-3}
Manhattan	7.09×10^{-2}	1.12×10^{-1}	5.22×10^{-3}
Euclidean	7.37×10^{-2}	1.17×10^{-1}	4.82×10^{-3}

Table 2: Performance of CNN with different signed distance maps

4.2 Activation function

We compare different activation functions to the ReLU. The idea was to test smoother functions, such as the Exponential Linear Unit (ELU) and Softplus. We also try Parametric ReLU (PReLU) which is a Leaky ReLU (LReLU) but the α of the formulae given below is a trainable parameter; in practice, LReLU is often used.

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leqslant 0 \end{cases}$$

$$\mathrm{LReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leqslant 0 \end{cases}$$

$$\mathrm{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{if } x \leqslant 0 \end{cases}$$

$$\mathrm{Softplus}(x) = \log(1 + e^x)$$

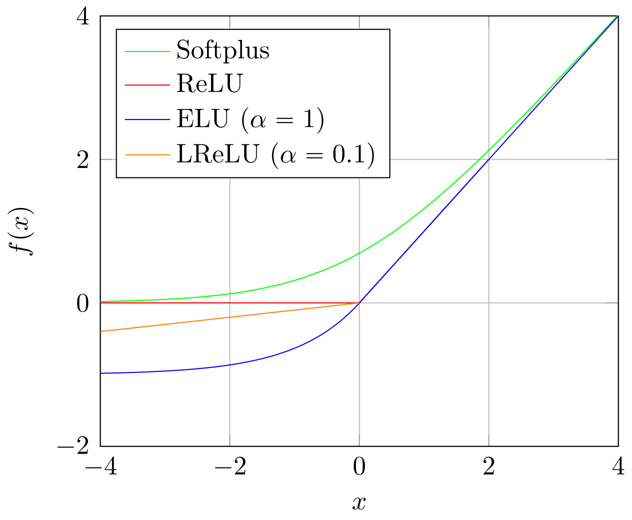

Figure 2: Different activation functions

In order to evaluate the performance, we trained the CNN on the same configuration as before but with the Manhattan distance as input. The complete configuration is described in Table 11, Experiment "Activation function". ELU is the activation function that provides the best results, Softplus also performed well. But for the rest of the experiments, we will stick to the ELU.

Activation function	ε_{Q}	$\varepsilon_{\max}$	$\varepsilon_{\mathrm{mean}}$
ReLU	7.09×10^{-2}	1.12×10^{-1}	5.22×10^{-3}
PLReLU	7.02×10^{-2}	1.10×10^{-1}	4.07×10^{-3}
ELU ($\alpha = 1$)	5.71×10^{-2}	8.86×10^{-2}	4.44×10^{-3}
Softmax	5.97×10^{-2}	9.38×10^{-2}	4.24×10^{-3}

Table 3: Performance of CNN with different activation functions

The impact of the activation function is highly visible on the convergence of metrics in validation. In figure 3, we compare the convergence of loss, $\varepsilon_{\max}$, and $\varepsilon_{\mathrm{mean}}$ in training and validation for the two CNNs with ELU and ReLU. With ReLU, a plateau is quickly reached for the validation. The training seems to have no impact on the convergence in validation, while with ELU, the convergence is clearly better.

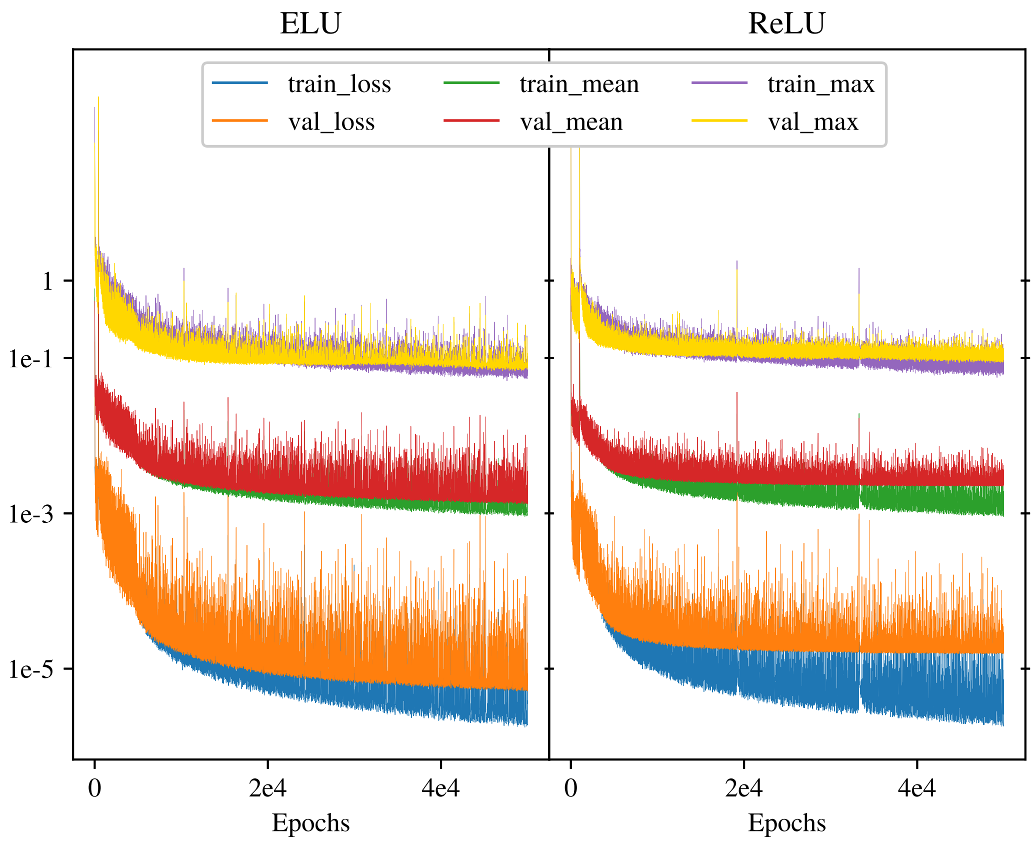

Figure 3: Difference in convergence using ELU and ReLU as activation functions

The CNN we now are trying to improve has an additional input with the signed distance map and has ELU as the activation function. This model has 2.99×10^5 trainable parameters.

5 Skip connection

One notorious issue of deep neural networks is the vanishing / exploding gradient, which is exacerbated by the number of layers causing degradation in accuracy. One way to overcome this problem is to use skip connections, this approach was first demonstrated in [6] with the ResNet network. Another way to address this issue is data normalization [7].

We tried to modify the architecture to see if it could improve the performance of our model. This section is divided into two parts, the first one concerns the data Partial_std and the second one the Full_std.In the first part, by mistake, the experiment was carried out using $\mathbb{E}[\varepsilon \circ \varepsilon]$ instead of $\text{MSE}(\gamma, \hat{\gamma})$ as a loss function. For time reasons, the impact of the loss function has only been evaluated on Full_std but not Partial_std.

5.1 First dataset

The first architecture we tried consisted in having blocks with two layers. Each block concatenates its input, i.e. the output of the previous block or layer, with the input of the previous block or layer. This architecture is described in Figure 4 with only two blocks. For 10 layers, with 4 blocks with 2 layers each, the model has 4.12×10^5 trainable parameter.

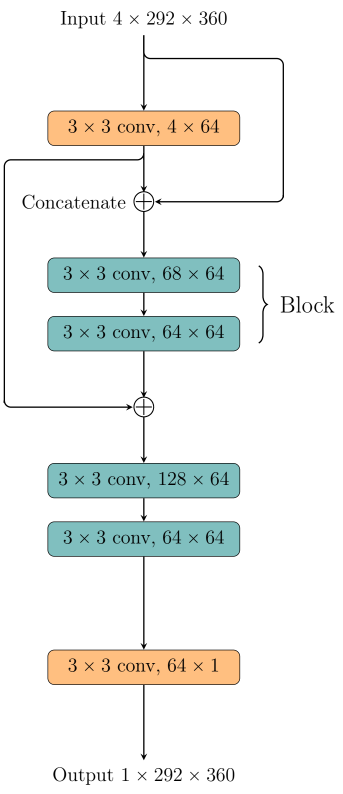
Input $4 \times 292 \times 360$
3×3 conv, 4×64
Concatenate
3×3 conv, 68×64
3×3 conv, 64×64
Block
3×3 conv, 128×64
3×3 conv, 64×64
3×3 conv, 64×1
Output $1 \times 292 \times 360$

Figure 4: Architecture representation with 2 blocks and skip connections

Skip connections	ε_Q	$\varepsilon_{\max}$	$\varepsilon_{\text{mean}}$
Without	5.71×10^{-2}	8.86×10^{-2}	4.44×10^{-3}
With	4.39×10^{-2}	6.64×10^{-2}	3.89×10^{-3}

Table 4: Performance of CNN with and without skip connections

5.2 Change of dataset

In this part, the data used for the training and the test is now the data set Full_std.

5.3 First architecture

We evaluate the performance of the previous model with different loss Experiment “Loss” and layers number Experiment “Layers”

Skip connections	loss	layers	ε_Q	$\varepsilon_{\max}$	$\varepsilon_{\text{mean}}$
Without	$\text{MSE}(\gamma, \hat{\gamma})$	10	2.99×10^{-2}	5.22×10^{-2}	4.07×10^{-3}
With	$\text{MSE}(\gamma, \hat{\gamma})$	10	2.83×10^{-2}	5.29×10^{-2}	4.32×10^{-3}
Without	$\mathbb{E}[\varepsilon \circ \varepsilon]$	10	2.25×10^{-2}	4.48×10^{-2}	3.72×10^{-3}
With	$\mathbb{E}[\varepsilon \circ \varepsilon]$	10	2.06×10^{-2}	4.24×10^{-2}	3.72×10^{-3}
With	$\mathbb{E}[\varepsilon \circ \varepsilon]$	16	2.06×10^{-2}	3.80×10^{-2}	3.55×10^{-3}

Table 5: Performance of CNN with and without skip connections with different loss functions and number of layers

The best results have been achieved with skip connections and $\mathbb{E}[\varepsilon \circ \varepsilon]$ as the loss function. In Figure 5, for the model with 10 layers, we can see that the worst performance, over 1%, is for pixels at a maximum distance of three from the coast. Furthermore, the further away the coast, the better the performance, which can be related to the distribution of ocean pixels by distance to coast shown in Figure 6.

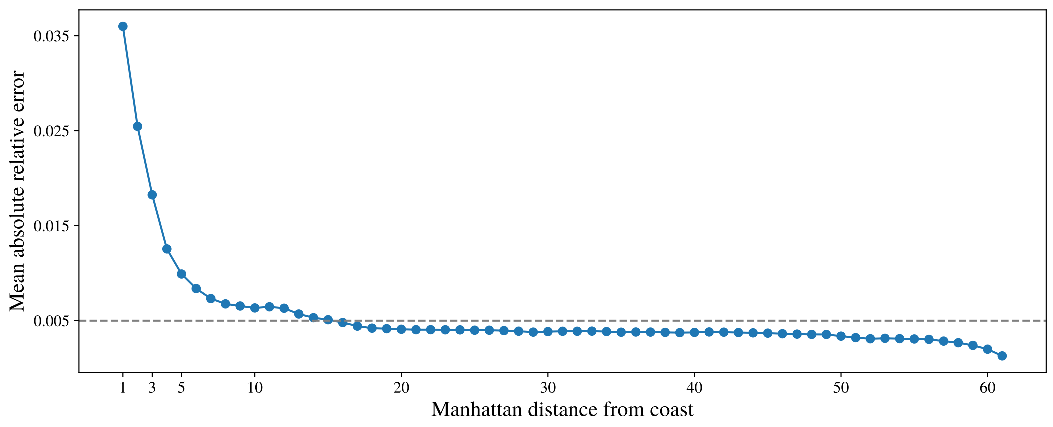

Figure 5: Mean absolute relative error with respect to the distance to the coast

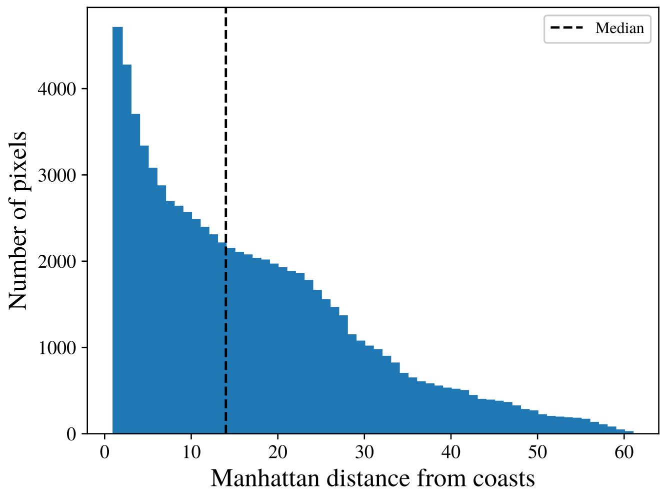

Figure 6: Distribution of ocean pixels by distance to coast

We see in Figure 7 the benefit of going to 16 layers is to reduce the maximum error for a distance of 1 to the coast. However, with 16 layers the model has 7.45×10^5 trainable parameters against 4.12×10^5 and the training duration is 15h30 against 13 hours.

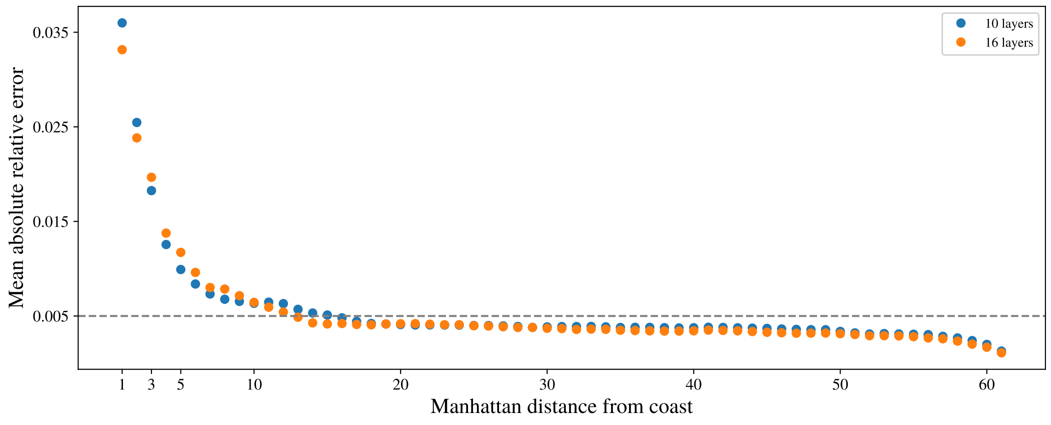

Figure 7: Comparison of the mean absolute relative error with respect to the distance to the coast between the best model with 10 and 16 layers

5.4 Second architecture

Afterward, we tried another type of block, instead of concatenating the data, each block adds up its input and output data as described in Figure 9. The CNN layout is shown in Figure 8. This architecture is inspired by the ResNet model [6]. This model is interesting because it has fewer trainable parameters than the first one, 2.99×10^5 against 4.12×10^5. This allows a faster training speed: 9 hours instead of 13 hours to reach 5×10^4 epochs.

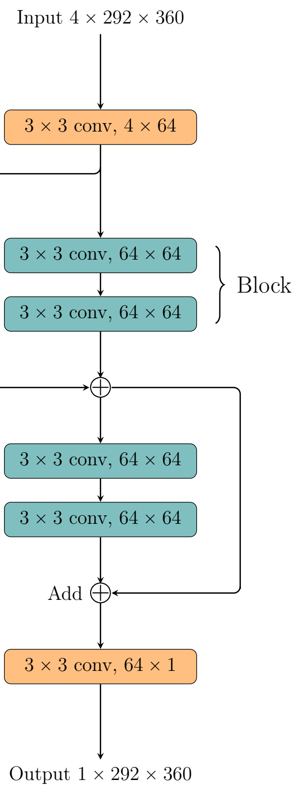

Figure 8: Architecture representation with 2 blocks and skip connections

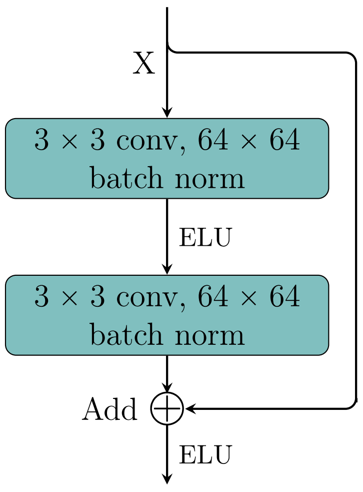

Figure 9: Block details

Skip connections	ε_Q	$\varepsilon_{\max}$	$\varepsilon_{\text{mean}}$
Concatenation (1st)	2.06×10^{-2}	4.24×10^{-2}	3.72×10^{-3}
Addition (2nd)	1.87×10^{-2}	4.27×10^{-2}	3.80×10^{-3}

Table 6: Comparison of the performance using two different types of skip connection

For the same number of layers, the results are better for ε_Q with the second architecture but slightly worse for the others metrics. We can see in Figure 10 that architecture 2 is better for pixels at distances 2 or 3 of the coast.

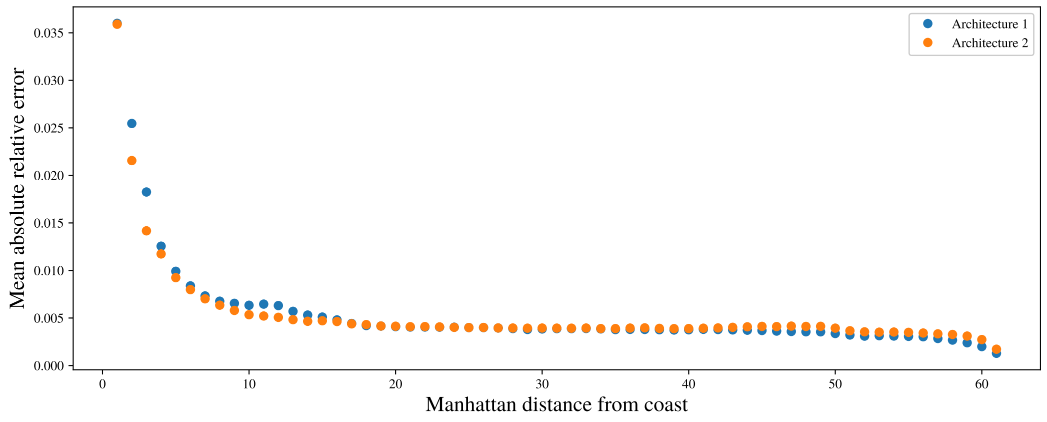

Figure 10: Comparison of the mean absolute relative error with respect to the distance to the coast between two types of skip connections

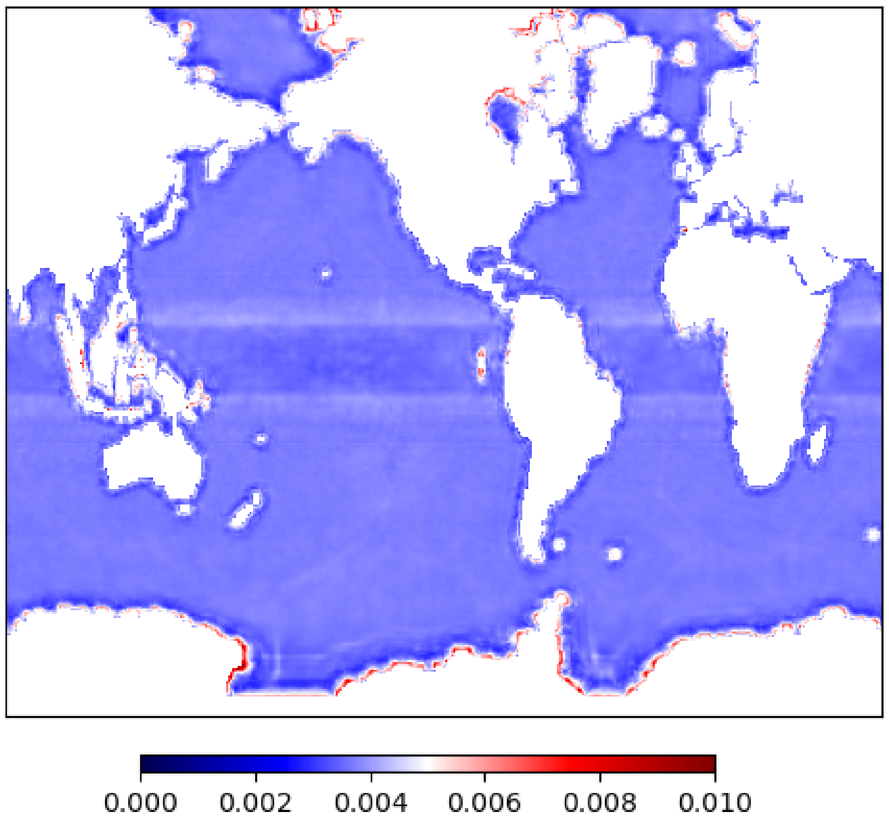

Figure 11: Map of $|\varepsilon|$ averaged on the test data set using the second architecture, Table 6 (second row)

6 Data augmentation

Data augmentation is commonly used for computer vision tasks with CNN, it aims to artificially increase the amount of data by transforming existing data. It helps to avoid overfitting by providing more data to the CNN [8]. Among the possible transformations we tried rotations and flips.

6.1 Method

As we have little data, we tried data augmentation with rotations and flips. Since the scaled diffusivity α_i (resp. α_j) is defined on the north (resp. east) of the cell, the scaled diffusivity needs to be changed according to the modification. For instance, with a vertical flip, if we look at a specific cell, the cell that was above is now below, meaning that the associated scaled diffusivity α_i should now be associated to the southern interface. To make the input channels consistent with each other, the field of α_i should be shifted south by one cell after the vertical flip, as shown on Figure 12. For the 90° and 270° rotations, α_i and

α_j are swapped as the axes are rotated. As the models' inputs are standardized, α_i and α_j must be properly restandardized.To restandardize, we used the normalization coefficients associated with $alpha_i$ and $alpha_j$ to destandardize the images A_i and A_j. Then we use the correct coefficients to standardise the images.

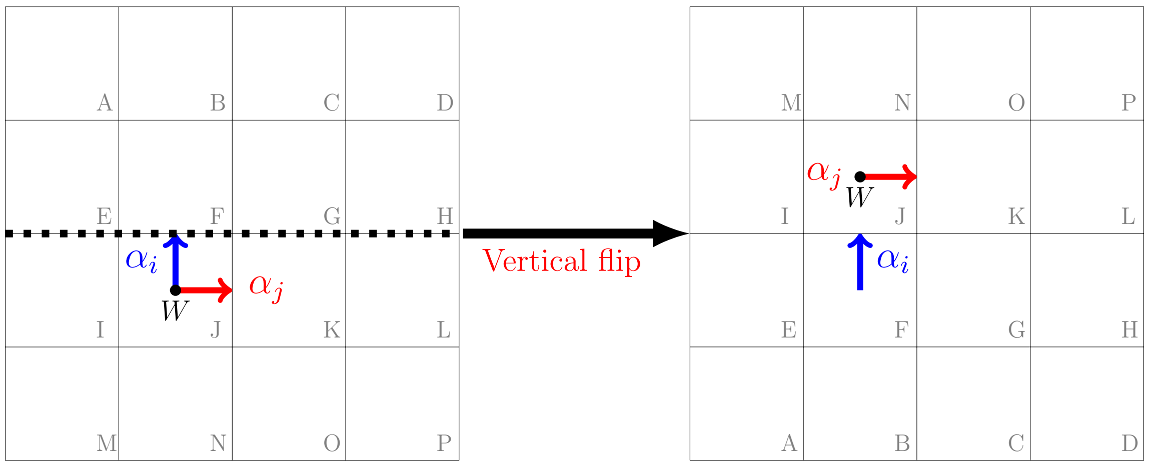

Figure 12: Representation of the vertical flip

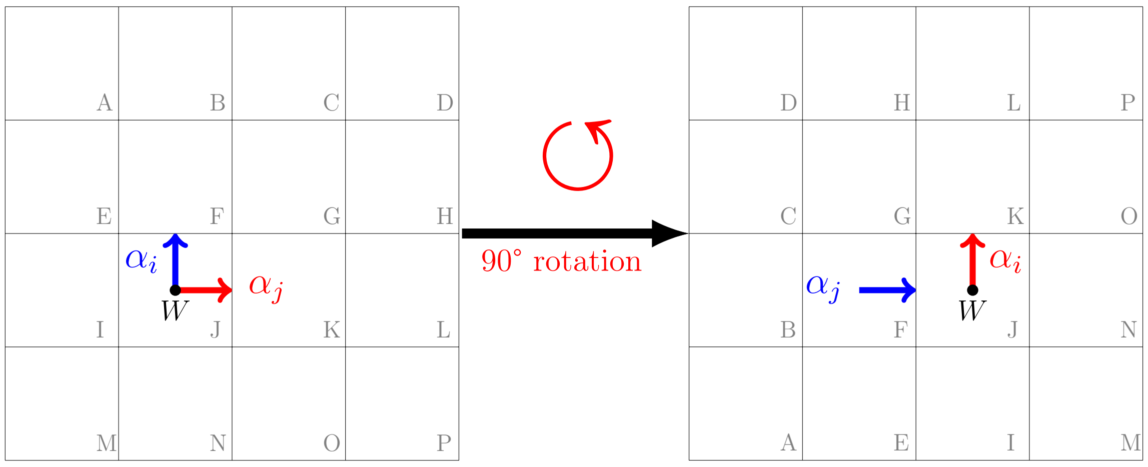

Figure 13: Representation of the 90° rotation

6.2 Results

We performed the training on the model with skip connections, $\mathbb{E}[\varepsilon \circ \varepsilon]$ as loss function, and 10 layers as described in Experiment "Augmentation". Augmentation is made on the data set Full_std.

Data set	ε_{Q}	$\varepsilon_{\max}$	$\varepsilon_{\mathrm{mean}}$
Original (10/1)	2.06×10^{-2}	4.24×10^{-2}	3.72×10^{-3}
Augmented with horizontal flips (20/1)	2.27×10^{-2}	4.78×10^{-2}	3.74×10^{-4}

Table 7: Impact of data augmentation.

The other transformations let to poorer results. The model converges quickly in validation, and the data augmentation does not impact the plateau of the validation curve.

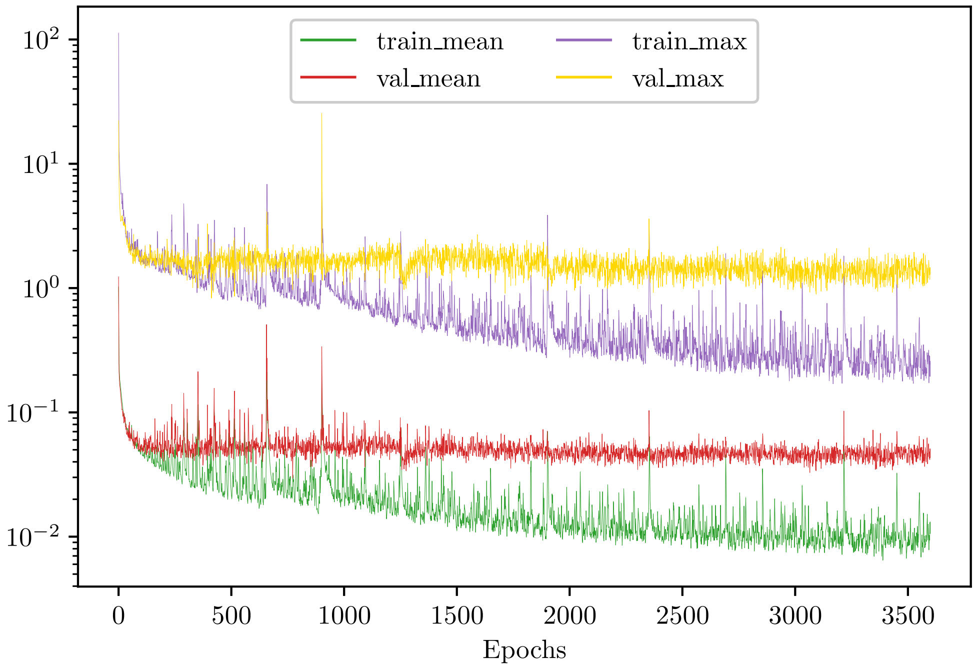

Figure 14: Difference in convergence in training and validation with augmented data with 90° rotation

7 Comparison with the randomization method

As the metrics of the randomization method with 10,000 samples are only avaible for the Python grid, we need to compute those metrics for the CNN on the Python grid. In order to do that, we train the best configuration (ELU + skip connections with addition), which we found by training on Nemovar grid, on the Python grid. The problem is slightly different between the Python and Nemovar grid, but this gives a good idea of the potential of the approach. The randomization method is described in [1]. As we can see in Table 8 the metrics are better by a factor of 2 for the CNN compared to the randomization. It should be noted that for the CNN the value of $\varepsilon_{\max}$ is lower than the one previously obtained on the Nemovar grid as the problem is easier. However, ε_{Q} remains similar.

Method	ε_{Q}	$\varepsilon_{\max}$	$\varepsilon_{\mathrm{mean}}$
Randomization (10^4)	5.4×10^{-2}	5.6×10^{-2}	1.1×10^{-2}
CNN	1.9×10^{-2}	2.8×10^{-2}	6.5×10^{-3}

Table 8: Comparison of randomization and deep learning

8 Conclusion and Perspectives

We achieved really good results with the CNN using skip connections and Exponential Linear Unit. The error is still high near the coasts but further tuning could be done to improve the performance. In the context of the experiments, the CNN performed better than the randomization method, so that our results provide a convincing proof of concept for the bidimensional case.

To go further with the CNN, we thought of doing the training on sections of the image instead of the whole image, which would allow to try out different strategies in the selection of the sections. It could be useful for importance sampling, but also for the computation of the training data, as only values for some pixels are needed. The major drawback of this approach is the computation cost, as there might be thousands of sections for a single image, making data loading a bottleneck.

All the experiments reported here are based on a 2D diffusion operator. Applying our results to operational data assimilation systems will require an extension of our work to the three-dimensional case. Possible ways to tackle the 3D problems would be to either use 3D convolution kernels or to apply the CNN to one layer and provide information on the other layers. The major issues we expect on this road are the computational cost of estimating the true normalization coefficients on a 3D grid, and the computational cost of the training.

References

[1] Anthony T. Weaver, Marcin Chrust, Benjamin Ménétrier, and Andrea Piacentini. An evaluation of methods for normalizing diffusion-based covariance operators in variational data assimilation. Quarterly Journal of the Royal Meteorological Society, 147 (734):289–320, 2021. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/qj.3918.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. URL http://www.deeplearningbook.org.

[3] Ding-Xuan Zhou. Universality of Deep Convolutional Neural Networks, July 2018. URL http://arxiv.org/abs/1805.10769.

[4] Philipp Petersen and Felix Voigtlaender. Equivalence of approximation by convolutional neural networks and fully-connected networks. January 2021. URL http://arxiv.org/abs/1809.00973.

[5] Olivier Goux. Correlation models based on diffusion. Internal note, 2022.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. URL http://arxiv.org/abs/1512.03385.

[7] Bouman Kak. Using skip connections to mitigate the problem of vanishing gradients, and using batch, instance, and layer normalizations for improved sgd in deep networks. Purdue University, April 2022. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/SkipConsAndBN.pdf.

[8] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. Journal of Big Data, 6(1):60, July 2019. URL https://doi.org/10.1186/s40537-019-0197-0.

Appendices

Appendix A Further results

Layers	Parameters	ε_{Q}	$\varepsilon_{\max}$	$\varepsilon_{\mathrm{mean}}$
5	1.5×10^5	3.1×10^{-2}	7.1×10^{-2}	1.2×10^{-3}
10	3.4×10^5	2.3×10^{-2}	4.8×10^{-2}	9.5×10^{-4}
15	5.2×10^5	4.4×10^{-2}	9.7×10^{-2}	1.8×10^{-3}
20	7.0×10^5	4.2×10^{-2}	9.1×10^{-2}	1.7×10^{-3}

Table 9: Performance of CNN for several numbers of layer on Python grid

Appendix B Reproduce the experiments

B.1 Packages and libraries

Python: 3.9.12
conda: 4.13.0
conda-build: 3.21.9

Package	Version	Channel
pytorch	1.11.0	pytorch
pytorch-lightning	1.6.4	conda-forge
torchvision	0.12.0	pytorch
wandb	0.12.17	conda-forge
numpy	1.22.3	

Table 10: Package versions

B.2 Configurations

Experiment	Distance map	Loss	layers	Channels	Activation functions	Data	Skip connection	Block/layer per block
Distance map	Var	MSE$(\gamma, \hat{\gamma})$	10	64	ReLU	Partial_std	None	
Activation function	Manhattan	MSE$(\gamma, \hat{\gamma})$	10	64	Var	Partial_std	None	
Skip connections	Manhattan	$\mathbb{E}[\varepsilon \circ \varepsilon]$	10	64	ELU	Partial_std	Var	4/2
Loss	Manhattan	Var	10	64	ELU	Full_std	With	4/2
Layers	Manhattan	$\mathbb{E}[\varepsilon \circ \varepsilon]$	Var	64	ELU	Full_std	With	Var /2
Augmentation	Manhattan	$\mathbb{E}[\varepsilon \circ \varepsilon]$	10	64	ELU	Var	With	4/2

Table 11: Configurations of the different experiments

Appendix C Run the project

C.1 Install the project

```
git clone https://github.com/FolkeKS/DL-normalization.git
```

C.2 Set up conda environment:

C.2.1 Create the environment (might be slow):

```
conda env create --file environment.yml
```

- If the installation is stuck on `Solving environment: |` try:

```
conda config --set channel_priority strict
```

- Revert with:

```
conda config - -set channel_priority true
```

C.2.2 Activate the environment:

```
bash conda activate DL-normalization
```

C.2.3 Setup project:

```
bash pip install -e .
```

C.3 Set up wandb for experiment tracking:

- Sign up at `https://wandb.ai/site` and log in
- Find your API-key at `https://wandb.ai/authorize`
- With your conda environment activated, run the following command and provide API-key

  ```
      wandb login
  ```

C.4 Train a model :

C.4.1 On a computer

```
python scripts/trainer.py fit --config  configs/demo.yaml
```

C.4.2 On a cluster using SLURM

- To continue a run after it has been stopped we set `ckpt_path: "results/wandb/cnn/36c5vu01/` (line 115). With `"cnn/"` the wandb project in which the run was created, and `"36c5vu01"` the 'id' of the run. We also set `version: 36c5vu01` (line 11) to make wandb continue the training in the run instead of creating a new run. The `global_step` variable will be reset, so the charts must be visualized with `epoch` or `step` as x-axis.

- It is also possible to choose how the best checkpoint is selected (lines 20-35), we choose the checkpoint that minimizes the loss in validation, but other choices are possible as choosing the one minimizing the quantile. However, we have not found how to use two strategies for the same run.

C.5 Modify the code

The model is loaded in scripts/trainer.py. Modifications are to be done in the config file but there are also modifications to make in the module loading the data in `src/data/dataset.py`:

- adjust the padding, by default the padding is 31 for the latitude and 28 for the longitude. The images are cropped on the fly, the dimension of the image taken by the CNN is $4 \times (292 + 2\ell) \times (360 + 2\ell)$ with ℓ the number of layers

- adding the sign distance map

For computational time, these modifications should be made to the data directly before starting the training.

```
    class DirDataset(Dataset):
    def __getitem__(self, i):
        idx = self.ids[i]
        X_files = glob.glob(os.path.join(self.X_dir, idx+'.*'))
        Y_files = glob.glob(os.path.join(self.Y_dir, idx+'_norm_coeffs.*'))
    #Load the input / true data
        X = torch.from_numpy(np.load(X_files[0])['arr_0']).float()
        Y = torch.from_numpy(np.load(Y_files[0])['arr_0']).float()
    #Load the distance map
        distance_map = np.load("data/python_sign_dist_map_std.npz")['arr_0']
        distance_map = torch.from_numpy(distance_map).float()
    #Crop the input data
        distance_map = transforms.CenterCrop([200, 360+2*10])(distance_map)
        X = transforms.CenterCrop([200, 360+2*10])(X)
    #Add the distance map
        X = torch.cat((X,torch.unsqueeze(distance_map, 0)),0)
        return X, \
            Y
```

C.6 Computing the metrics on the test data set

As explained in the report, we used one data set for the python data and two data sets **Partial_std** = **finaldata** and **Full_std** = **newdata** for the Nemovar grid. However, for the **Partial_std** data set the training data were standardized using 10 samples while the test data were standardized using 180 samples. So, to compute the metrics the test data had to be destandardized and then restandardized. For **Full_std** all the data are standarized using the 10 training samples.

We compute the mean and standard deviation of the mean/max/quantile 99,99% of the absolute relative error over the test dataset. We save the tensor of the relative error for each sample and save an image of the mean of the relative error. The code and results are in the repository `results/test_metrics/` each repository corresponds to a data set. To run the computation of the metrics, run the python scripts from the root directory of the project. The configuration of the runs is added to a list `model_params`, each element of this list is a list with the index corresponding to:

0. Boolean: compute or not the metrics
1. Boolean: if the run uses the sign distance map
2. Numpy array: the sign distance map
3. Int: number of layers
4. Class: the model loaded with the correct src

C.7 Compute the sign distance map

The sign distance map is computed in the notebook `notebooks/Distance_map.ipynb`. We use the method `scipy.ndimage.distance_transform_bf` to compute the distance (doc link).

C.8 Perform the data augmentation

The script to use is `src/data/augmentation.py` and the combinations of transformations are tested in the notebook `notebooks/test_augmentation.ipynb`. As the sign distance map is added to the input data during the augmentation, it is important to not add it an other time as described in Section C.5.

List of Figures

List of Tables