Research project (PRe)

Specialty : STIC
Academic year : 2020/2021

Optimization of dynamic neural field parameters with pytorch

Privacy Notice

non-confidential report

Author : Mahdi Cheikhrouhou

Promotion : 2022

Tutor ENSTA Paris : M. Gianni Franchi

Tutor CentraleSupelec : M. Jeremy Fix

Internship period: 17/05/2021 - 20/08/2021

Host organization: CentraleSupélec campus of Metz

Address: 2 Rue Edouard Belin, 57070 Metz, France

Abstract

In this report we will discuss the approach that we took to optimize the parameters of the dynamic neural field by presenting the functions that we used as well as two different scenarios to test our approach and their results . We used *Pytorch* library with python to optimize these parameters.

Résumé

Dans ce rapport, nous discuterons de l'approche que nous avons adoptée pour optimiser les paramètres du champ neuronal dynamique en présentant les fonctions que nous avons utilisées ainsi que deux scénarios différents pour tester notre approche et les résultats trouvés. Nous avons utilisé la bibliothèque *Pytorch* avec python pour optimiser ces paramètres.

Acknowledgment

I would like to thank Mr. Jeremy Fix my internship supervisor , Mr. Gianni Franchi my ENSTA Paris Tutor for their help and their useful tips and advices throughout the internship. It's my first research internship and i learned many technical as well as soft skills from it. I would like to also thank Centrale Supélec for the equipments that they gave at the start of the internship and for giving me the chance to work on their GPU servers.

Contents

List of Figures

Chapter 1

Dynamic neural field

1.1 Introduction

Dynamic Neural Fields (DNF) is a mathematical model that simulates the state of some neurons over the time. This domain is the origin of multiple research papers as in [1]. These neural fields are used in a variety of applications like medicine and robotics as mentioned in the paper [4]. The model takes into considerations the input for each neuron and the relationship between them. The response of dynamic neural fields is highly dependent on its parameters and tuning these parameters is usually done manually and still pretty hard. This work aims at proposing a library for optimizing dynamic neural fields. This work focuses on differential programming, using mainly gradient descent, relying on the recent development of highly efficient differential programming toolbox such as pytorch. Other approaches for optimizing dynamic neural fields have been explored in the literature like swarm particle optimization [5].

1.2 Mathematical model

The equation of a dynamic neural field describes the evolution of the potential $u(x,t)$ of a neural population. x is the position (of the neuron) in the field, it can be $1d$ or $2d$ it depends on the scenario.

$$\tau \frac{\partial u}{\partial t}(x,t) = -u(x,t) + \int_y w(x-y) f(u(y,t)) + I(x,t) + h$$

The equation can be discretised in space and in time with the forward Euler approximation method, so we obtain this new equation that we can simulate :

$$u(x, t+\delta) = u(x,t) + \delta \Delta u(x,t)$$

$$\Delta u(x,t) = \frac{1}{\tau}\Big(-u(x,t) + \sum_y w(x-y) f(u(y,t)) + I(x,t) + h \Big)$$

The meaning of each term in the equation :

- f: is the activation function or transfer function. We typically use *Sigmoid* or *Relu* which we will explore more in 1.3.

- I: is the input of the neurons which depends on the scenario. We will talk more about scenarios later.
- h: is the resting potential
- w: is the lateral weights which represents the relationship between the neurons and how they affect one another which we will explore more in 1.4.

1.3 Transfer functions

1.3.1 Sigmoid

Sigmoid is a transfer functions which converts the space of real numbers into the interval $[0 : 1]$. In our work we consider this parametrization of the sigmoid :

$$Sig_{\alpha,x_0,\beta}(x) = \frac{1}{1 + \beta e^{-\alpha(x-x_0)}}$$

where the parameters α , x_0 and β are trainable.

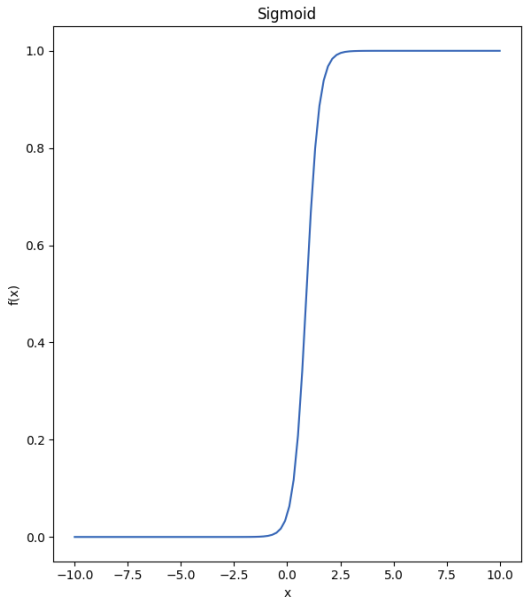


Figure 1.1: Graph of the sigmoid function

1.3.2 Relu

Relu is a transfer function that converts the space of real numbers into the space of positive real numbers as follows :

$$relu_{x_0}(x) = max(x - x_0, 0)$$

where the parameter x_0 is trainable.

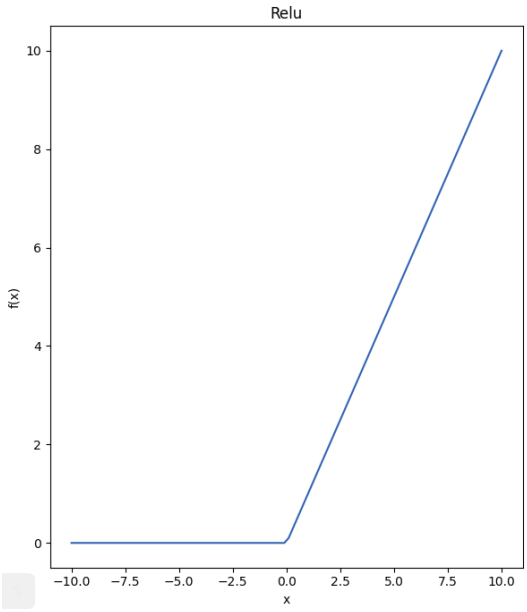


Figure 1.2: Graph of the relu function

1.3.3 Initialization

In all of the word we used the *sigmoid* transfer function and with some multiple experiences we found out the best initialization intervals which are :

Parameter name	initial range (uniform)
α	$[3.3 : 3.5]$
β	$[1 : 1.5]$
x_0	$[0.5 : 1]$

1.4 Weights functions

To define the weights functions we need first to define some base functions .

1.4.1 Gaussian

The gaussian function we use in our work is in the form :

$$gauss_{\alpha,\sigma}(x) : \alpha.exp(-x^2/(2\sigma^2))$$

where α and σ are trainable parameters. The gaussian function is the now usually used in works on dynamic neural fields.

1.4.2 Exponential

The exponential function we use in our work is in the form :

$$exp_{\alpha,\sigma}(x) : \alpha.exp(-4|x|/(2\sigma^2))$$

where α and σ are trainable parameters. The exponential weight function has been introduced in [3]. It can be efficiently implemented in hardware contrary to the gaussian function.

1.4.3 Linear

The linear function we use in our work is in the form :

$$lin_{\alpha,\sigma}(x) : \alpha\Big[1 - \frac{|x|}{2\sigma}\Big]^+$$

where α and σ are trainable parameters and $[x]^+ = max(x, 0)$.

The linear kernel as well as the heaviside kernel considered in the next section have been considered in [2]. These are very simple kernels that can be very efficiently implemented in hardware and software that have been shown to preserve the important behaviors experimented with gaussian kernels.

1.4.4 Heaviside

The heaviside function we use in our work is in the form :

$$Heav_{\alpha,\sigma}(x) = \alpha.H(\sigma - |x|)$$

where α and σ are trainable parameters and $H(x) = 1$ if $x > 0$ otherwise $H(x) = 0$.

1.4.5 Types of weights functions

We used 5 different types of weights functions. Let's consider 4 real numbers σ_+ ,σ_-, α_+ and α_- such that $\sigma_+ \leq \sigma_-$ and $\alpha_+ \geq \alpha_-$. To ensure these constraints we define them in the following way :

- $\alpha_+ = exp(x_\alpha)$
- $\alpha_- = sigmoid(k_\alpha) * \alpha_+$
- $\sigma_- = exp(x_\sigma)$
- $\sigma_+ = sigmoid(k_\sigma) * \sigma_-$

With $x_\alpha, k_\alpha, x_\sigma, k_\sigma$ are in R . In this way we guarantee that $0 \leq \alpha_- \leq \alpha_+$ and $0 \leq \sigma_+ \leq \sigma_-$

We define the weights functions as follows :

- Difference of Gaussians (DOG): $dog(x) = gauss_{\alpha_+,\sigma_+}(x) - gauss_{\alpha_-,\sigma_-}(x)$
- Difference of Exponentials (DOE): $doe(x) = exp_{\alpha_+,\sigma_+}(x) - exp_{\alpha_-,\sigma_-}(x)$
- Difference of Linears (DOL): $dol(x) = lin_{\alpha_+,\sigma_+}(x) - lin_{\alpha_-,\sigma_-}(x)$
- Rectangular Mexican Hat (step): $step(x) = heav_{\alpha_+,\sigma_+}(x) - heav_{\alpha_-,\sigma_-}(x)$

- Difference of Gaussian and constant (DGC): $dgc(x) = gauss_{\alpha_+,\sigma_+}(x) - \alpha_-$

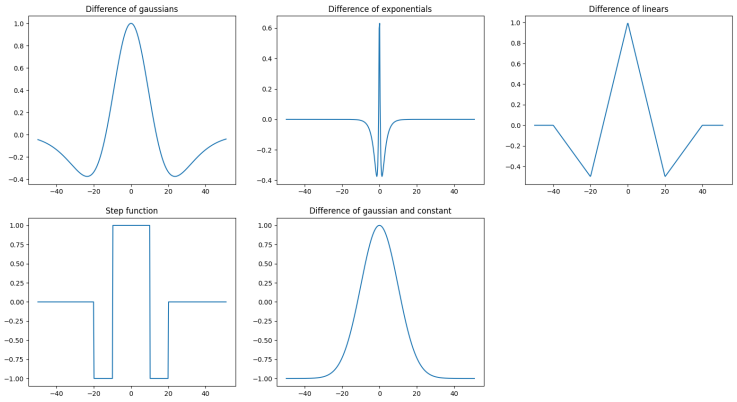


Figure 1.3: Weights functions

1.4.6 Initialization

The weights parameters are initialized in the following way :

Parameter name	initial range (uniform)
x_α	$[-1 : 0]$
k_α	$[-0.5 : 0.5]$
x_σ	$[-3 : -2]$
k_σ	$[-0.5 : 0.5]$

which gives us the corresponding parameters :

Parameter name	initial range
α_+	$[0.36 : 1]$
α_-	$[0.37\alpha_+ : 0.62\alpha_+]$
σ_-	$[0.049 : 0.13]$
σ_+	$[0.37\sigma_- : 0.62\sigma_-]$

1.5 Model performance

Depending on the values of the parameters, the dynamic neural fields exhibit qualitatively different responses. In our work, we want to optimize these parameters so that the response matches a desired response. Our aim is to maximise the target bumps from the input to the output. To do so , we define our cost function introduced in [5] as a template that the models outputs should follow. To measure the performance of the model with the current parameters we use the $lower\_bound - upper\_bound$ loss , which penalizes the parts of the response that are out of the area contoured by the defined lower_bound and upper_bound which are defined by :

$$s(x) = \frac{1}{1 + exp(-15(x - 0.5))}$$

$$lb(x) = H\Big(\pi/4 - \frac{d(x,x_0)}{\sigma - \Delta\sigma}\Big)\Big[(A-\Delta A)cos\Big(\frac{\pi}{4}\frac{d(x,x_0)}{\sigma-\Delta\sigma}\Big)\Big]^+$$

$$ub(x) = s((A+\Delta A)exp(-\frac{d(x,x_0)^2}{2(\sigma+\Delta\sigma)^2})$$

where H is the heaviside function.

The expression of the cost function then becomes :

$$cost = \sum_x b(tf(x), x)^2$$

$$b(y,x) = \begin{cases} ub(x) - y & \text{if } y \geq ub(x) \\ y - lb(x) & \text{if } y \leq lb(x) \\ 0 & \text{otherwise} \end{cases}$$

where tf denotes the transfer function.

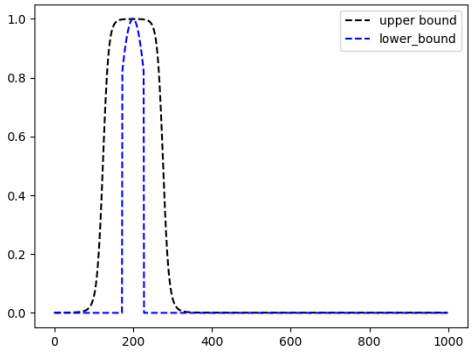


Figure 1.4: Graph of the lower bound and upper bound function

1.6 Scenarios

As in [5], we will optimize a set of parameters for 2 different scenarios which are :

- Competition scenario
- Working memory scenario

Chapter 2

Competition scenario

2.1 Definition

The competition scenario consists of multiple bumps in the input. Only one of the bumps has a higher amplitude than the others which is expected to be selected by our model. The rest of the bumps have the same amplitude. And that's where the name competition comes from , so in fact there is a competition between the bumps but only the one with the highest amplitude will win and the others will diminish.

Our objective is to train the parameters so given such input we will have only the main bump in the output. To make things more challenging , we choose the weak amplitude from $0.4, 0.5$ and the strong amplitude is greater than the weak amplitude by 0.5. Adding this kind of probability makes our parameters more robust and applicable to more examples.

2.2 Implementation details

We implemented the scenario as subclass of *torch.utils.data.IterableDataset* which is a *dataset* module based on *pytorch* and *torch tensors*. To do so, we split the time steps into 3 phases (figure 2.1) which are :

- Rising phase : during this phase the amplitude for each bump increases linearly from 0 to reach it's maximum.
- maximum phase : during this phase the amplitude of each bump stays the same for the whole period.
- Falling phase : during this phase the amplitude for each bump decreases linearly from the maximum to reach 0.

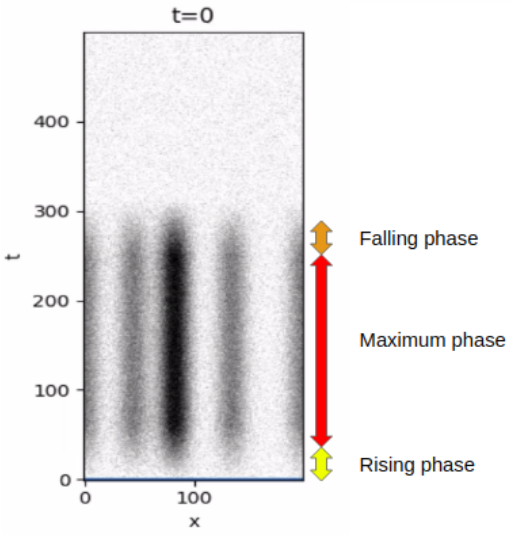


Figure 2.1: Visualization of the different phases

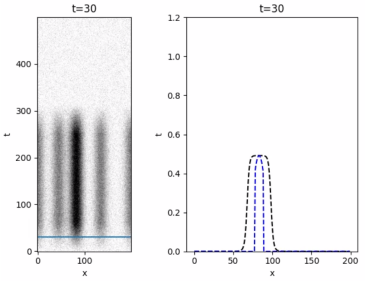


Figure 2.2: The middle of the rising phase en 1d

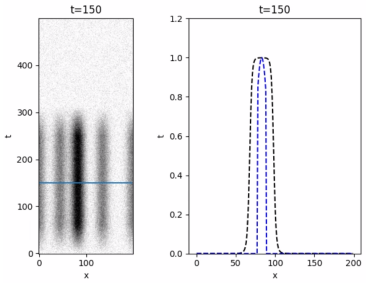


Figure 2.3: The middle of the maximum phase en 1d

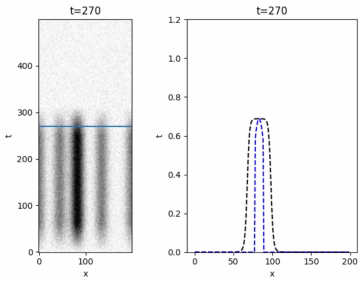


Figure 2.4: The middle of the falling phase en 1d

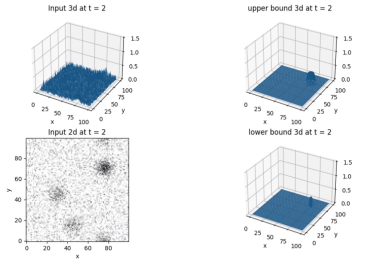


Figure 2.5: The middle of the rising phase en 2d

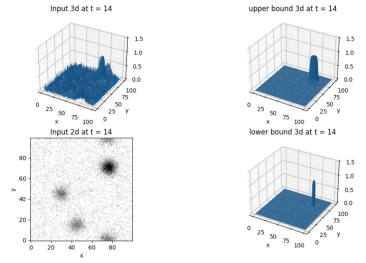


Figure 2.6: The middle of the maximum phase en 2d

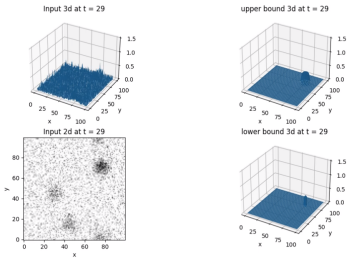


Figure 2.7: The middle of the falling phase en 2d

In the figures 2.2, 2.3 and 2.4 we can see 4 bumps where only one with higher amplitude (the darkest one) . The right half of each figure represents the lower bound and upper bound of the cost function (lower bound represented in blue and upper bound represented in black). With this we can be sure that the peak expected from the cost function is aligned perfectly with the main bump. Another thing to

notice is that in the example shown , the field is cyclic that's why it seems that exists 4 distractors but in fact the left most and right most ones are the same bump. We make the field cyclic to simulate an infinite size environment. The figures 2.5, 2.6 and 2.7 are for the 2d implementation of the competition scenario. Each figure is divided into 4 sub-figures which are :

- top-left : The input represented in 3d
- bottom-left : The input represented as a gray scale image
- top-right : The upper bound of the cost function
- bottom-right : The lower bound of the cost function

2.3 Training details

To train the model we used the following parameters:

Variable name	Description
size	the size of the field
num_steps	number of time steps of a single simulation
cyclic	Boolean variable to make the field cyclic or not
lr	the initial learning rate
rand	Indicates to start from random initialization
batch_size	The size of each batch
nb_batchs	the number of training batches

The values for the parameters for each dimension are :

Variable name	Value for 1d	Value for 2d
size	100	30x30
num_steps	100	80
cyclic	True	True
dim	1	2
lr	0.005	0.005
batch_size	32	32
nb_batchs	1000	1000
transfer_function	Sigmoid	Sigmoid
weights_function	DOG	DOG

2.4 Results analysis

After optimizing the parameters we have a couple of results to explore.

2.4.1 Loss

The graph of the loss over the time (figure 2.8 and figure 2.9) gives us an idea about how good are the parameters we trained and how much time it needed to reach the optimum. In this case , it is clearly that our model found the perfect parameters hence the loss is very close to zero. In addition , between 60 and 80 epochs for the competition 1d and between 250 and 270 for the competition 2d, were enough to find those parameters.

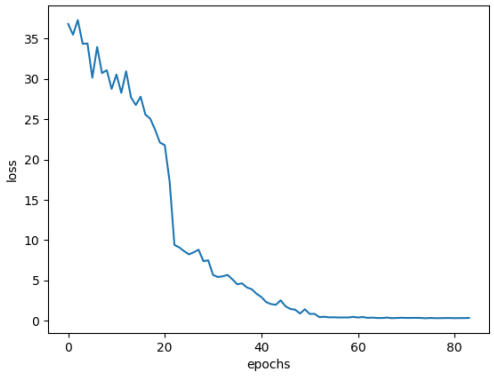


Figure 2.8: Loss for competition 1d

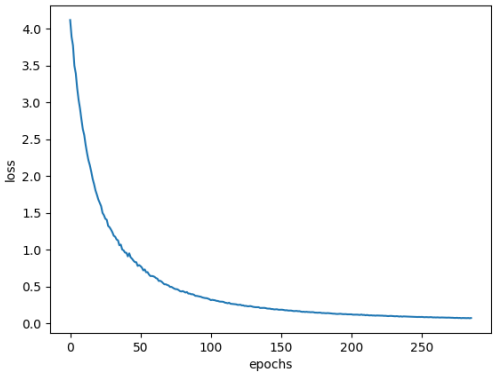


Figure 2.9: Loss for competition 2d

2.4.2 Parameters

The optimum parameters values for competition 1d are as follow :
Weight function :

Parameter name	optimum value
α_+	0.2804
α_-	0.1516
σ_-	0.2609
σ_-	0.0857

Transfer function :

Parameter name	optimum value
α	3.0143
β	1.3455
x_0	1.1619

The optimum parameters values for competition 2d are as follow :
Weight function :

Parameter name	optimum value
α_+	0.8804
α_-	0.5743
σ_-	0.1315
σ_+	0.0656

Transfer function :

Parameter name	optimum value
α	3.9069
β	1.5023
x_0	1.0660

2.4.3 Weights

This subsection is consecrated only to analyse the weights that we found after the optimization. The figures 2.10 and 2.11 represents the weights for the competition 1d and competition 2d centered in points (0) and (0,0) respectively. To achieve the competition scenario we need only one bump in the output that means all the others will diminish and will be removed . This characteristic is ensured by the form of the weights since they have local excitation but a global inhibition for all the rest of the field.

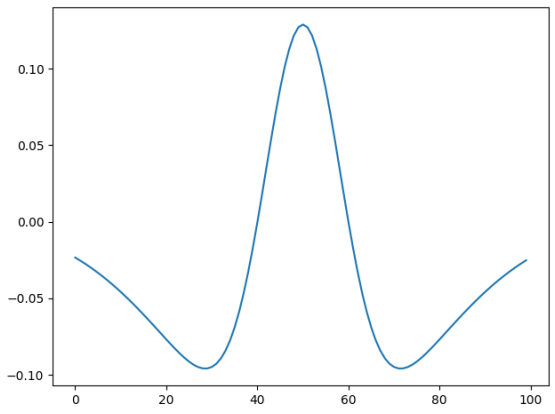


Figure 2.10: Weights for competition 1d

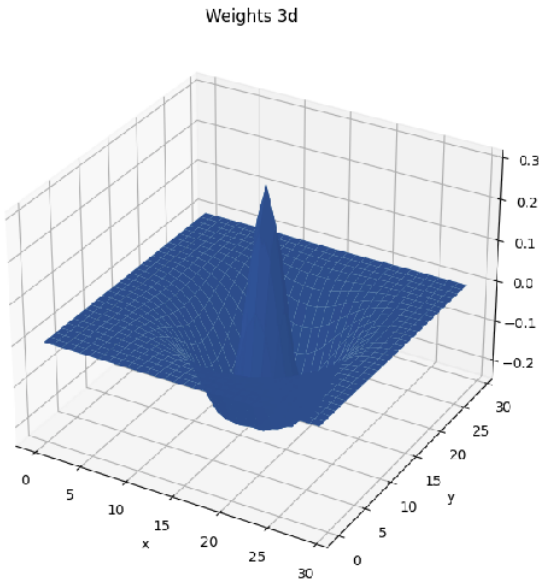


Figure 2.11: Weights for competition 2d

2.4.4 Output

One more thing to explore is the output results compared to the input and the expected results.

The figure 2.12 represents the results for the competition 1d. The left figure is the input , the middle image is the output and the right image is cost function (*lowerbound* in orange , *upperbound* in black), $f(u(x,t))$ in red and $u(x,t)$ in blue. From the output figure , we can clearly see that the model worked as expected because $f(u(x,t))$ fits almost perfectly between the *lowerbound* and the *upperbound*.

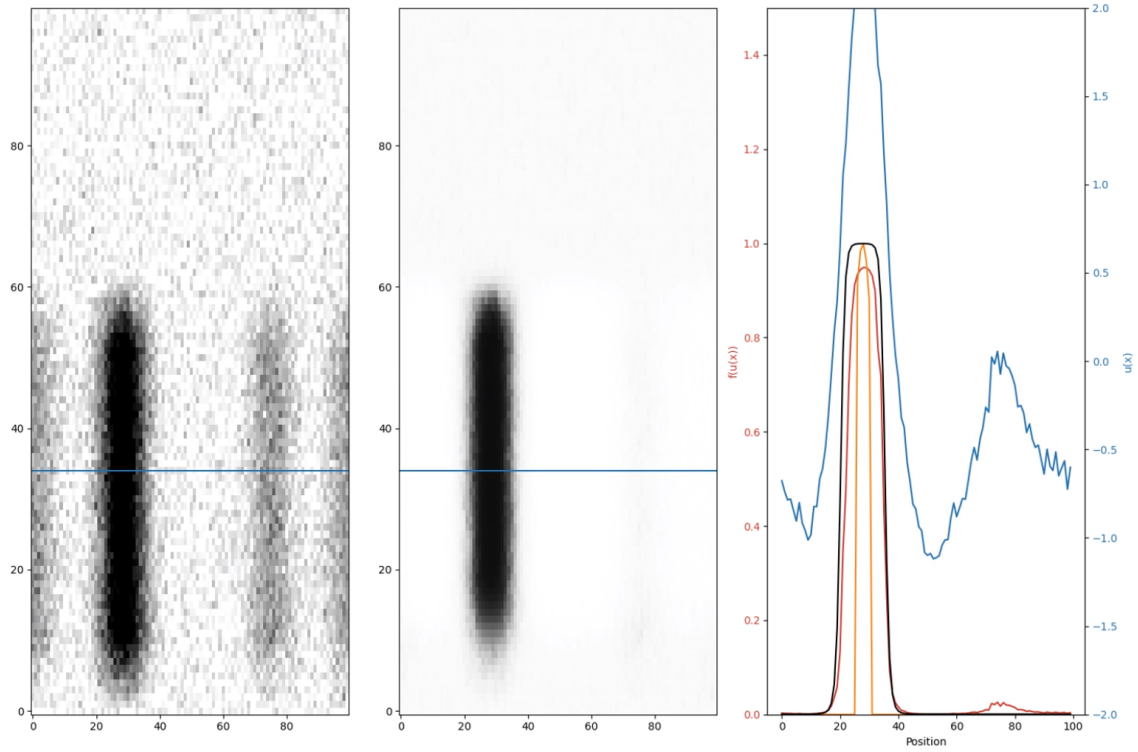


Figure 2.12: Left: input , middle : output, right : cost function for the competition 1d

For the competition 2d , we represent the output in different ways (figure 2.13). First , in the right we find the input in 3d and 2d. Second , the output is presented in the middle in 2d and 3d form. We can see the model optimized the parameters well if we compare the 3d or the 2d representation to the expected result of the input (the bump with the highest amplitude). Finally, in the right we see the part that exceeds the upper bound in the top and the part that exceeds the lower bound in the bottom. We can notice that the lower bound and the upper bound are almost perfectly respected , so our result is very good.

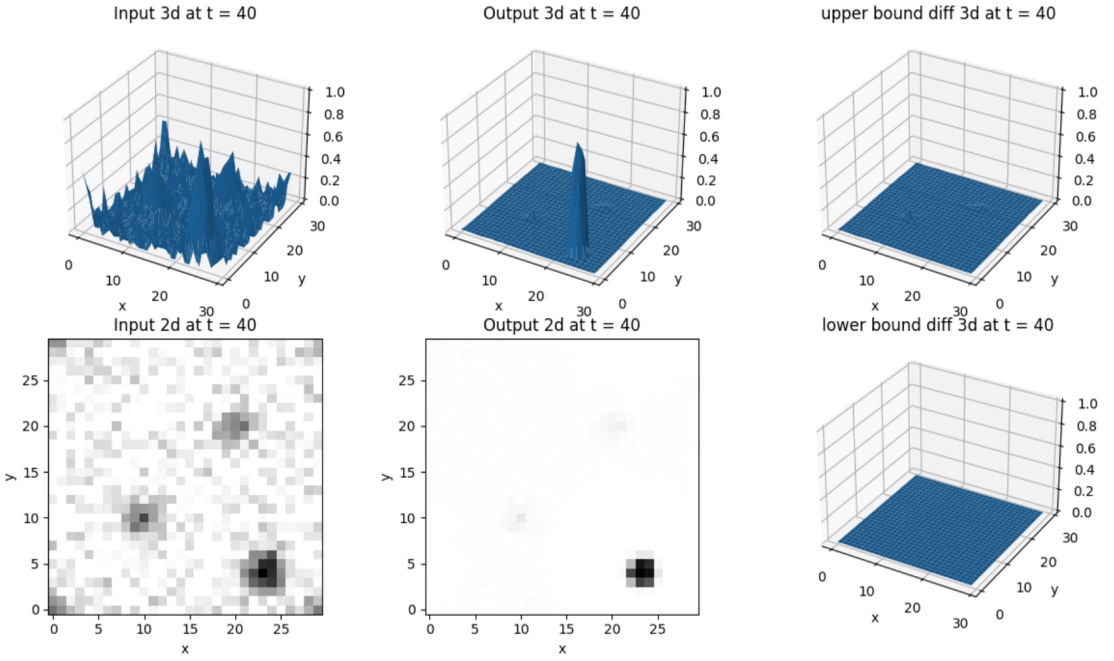


Figure 2.13: top-left : Input in 3d , bottom-left : Input in 2d, top-middle : output in 3d, bottom-middle : output in 2d, top-right : difference of f and upper bound , bottom-right : difference of f and lower bound

Chapter 3

Working memory scenario

3.1 Definition

The working memory scenario consists of multiple bumps to track . All the bumps have the same amplitude but each one has a trigger moment from when the memory work begins.

Our objective is to train the parameters so for each bump the output is null before the trigger moment and maximum after the trigger moment. That's why we call it memory , it's like memorizing everything after the trigger moment.

3.2 Implementation details

We implemented the scenario as a subclass of *torch.utils.data.Dataset* which is a dataset module based on *pytorch* and *torchtensor*. We split the time steps into 4 phases which are :

- Pre trigger Phase : The amplitude of the bump is constant and low
- Trigger phase : The amplitude augments rapidly and linearly to the maximum, stays constant for some time steps and then decreases rapidly and linearly to the low amplitude
- Post trigger phase : The amplitude of the bump is constant and low
- Deviation phase : the bump starts deviating to the right till the end.

We added the deviation phase to make the problem more challenging and interesting but in the same time more complex. To memorise a bump , it's enough to have strong lateral excitatory connections. But if they are way too strong , there is a risk that the memory will self-power regardless of the input. Having a bump in memory which is updated with the input therefore brings a stronger constraint on the space of possible parameters

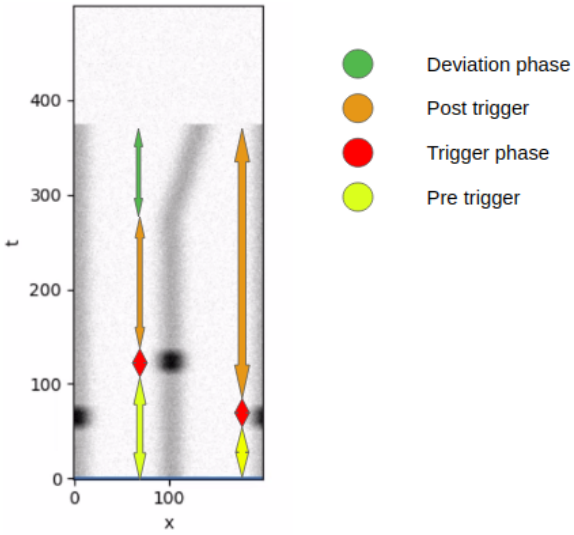


Figure 3.1: Visualization of the different phases

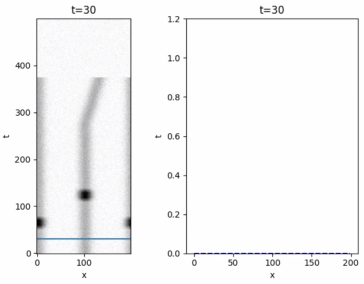


Figure 3.2: The middle of the pre trigger phase en 1d

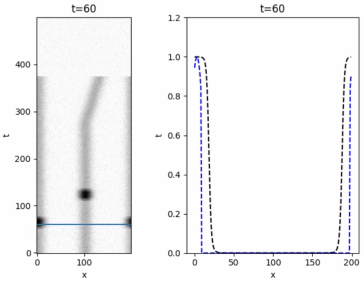


Figure 3.3: The middle of the first trigger phase en 1d

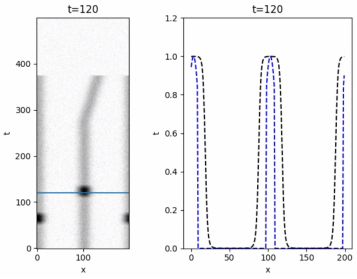


Figure 3.4: The middle of the second trigger phase en 1d

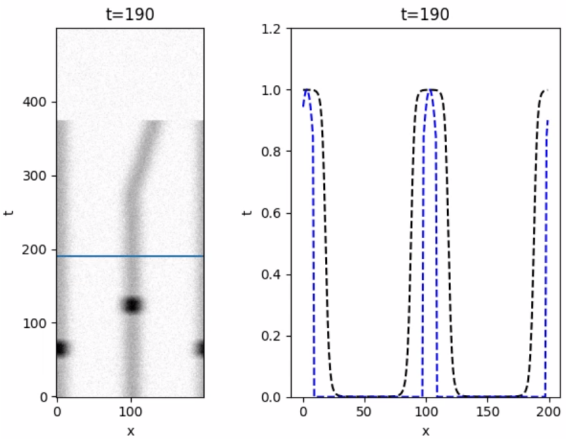


Figure 3.5: The middle of the post trigger phase en 1d

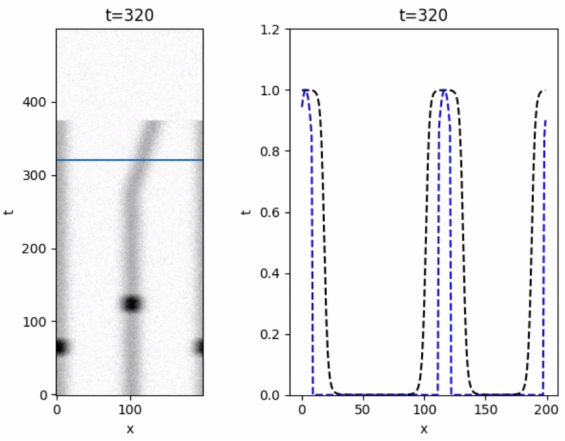


Figure 3.6: The middle of the deviation phase en 1d

The figures 3.2, 3.3, 3.4, 3.5 and 3.6 represents the 4 phases that we already

talked about. The left images represent the input at different time steps and the right images represents the lower bound upper bound loss template (lower bound in blue and upper bound in black). Note that there is 2 templates in the cost function that corresponds to the number of bumps in the input. Just one more thing to notice , is the loss template for the middle bump also changes position with the deviation.

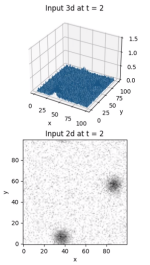


Figure 3.7: The middle of the pre trigger phase en 2d

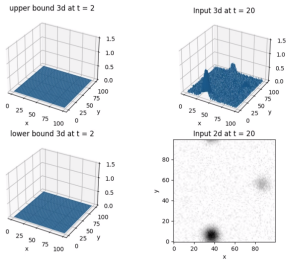


Figure 3.8: The middle of the first trigger phase en 2d

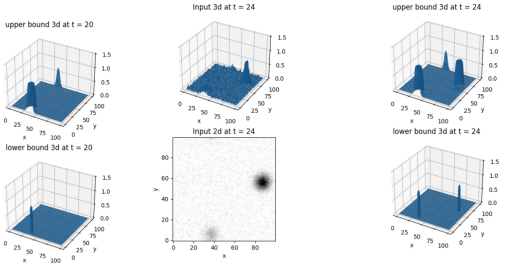


Figure 3.9: The middle of the second trigger phase en 2d

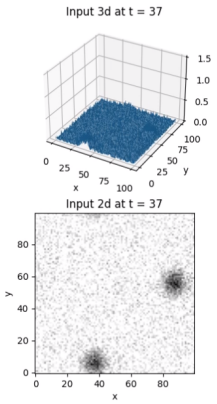


Figure 3.10: The middle of the post trigger phase en 2d

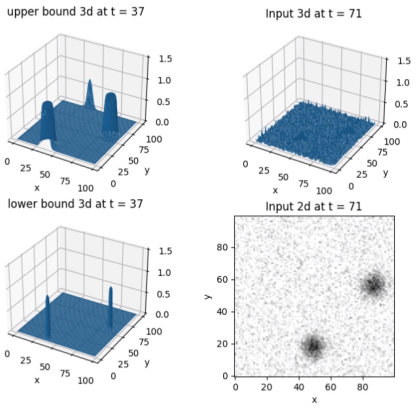
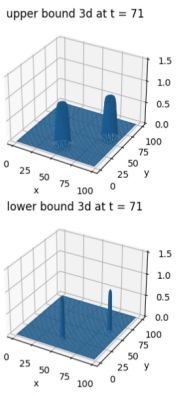


Figure 3.11: The middle of the deviation phase en 2d



The figures 3.7, 3.8, 3.9, 3.10 and 3.6 represents the working memory phases but in a 2d field. Each figure composed of 4 graphs :

- Top left : the input in a 3d plot
- Bottom left : the input in a 2d gray scale image
- top right : the upper bound of the cost function
- bottom right : the lower bound of the cost function

3.3 Training details

To optimize the parameters we used the following parameters :

Variable name	Description
size	the size of the field
num_steps	number of time steps of a single simulation
cyclic	Boolean variable to make the field cyclic or not
dim	the dimension of the field 1d or 2d
lr	the initial learning rate
batch_size	The size of each batch
nb_batchs	the number of training batches

The values for the parameters for each dimension are :

Variable name	Value for 1d	Value for 2d
size	100	30x30
num_steps	100	80
cyclic	True	True
dim	1	2
lr	0.005	0.005
batch_size	32	32
nb_batchs	1000	1000
transfer_function	Sigmoid	Sigmoid
weights_function	DOG	DOG

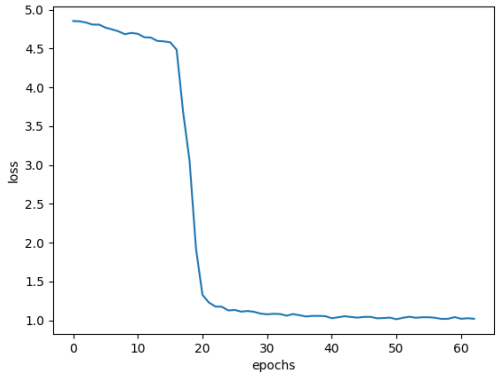
The number of batches for the 2d competition is low because of the huge time of computation of a single batch.

3.4 Results analysis

After optimizing the parameters , we have a couple of results to explore :

3.4.1 Loss

The loss over the time represented in the figure 3.12 and the figure 3.13 shows that the loss decreased to reach a stationary phase at the end at ≈ 30 epochs. We can't actually know the performance of the model since for now we don't know what is a low loss for this scenario to compare to. So to measure the performance we will need to check the subsection 3.4.4 .



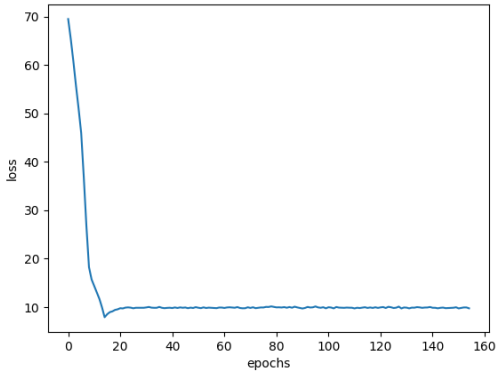


Figure 3.12: Loss for working memory 1dFigure 3.13: Loss for working memory 2d

3.4.2 Parameters

The trained parameters for working memory 1d are as follow : Weight function :

Parameter name	optimum value
α_+	0.6412
α_-	0.3324
σ_-	0.0737
σ_+	0.0418

Transfer function :

Parameter name	optimum value
α	3.9069
β	1.5023
x_0	1.0660

3.4.3 Weights

The shape of the weighs in the figures 3.14 and 3.15 are as expected since we need to have local excitation to detect the trigger, local inhibition to make the bump local so it will not spread over the whole grid.

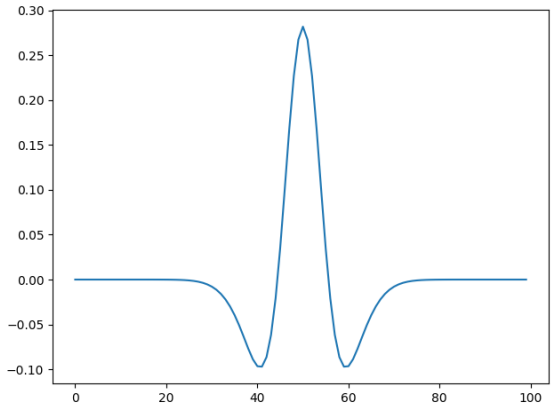


Figure 3.14: Weights for working memory 1d

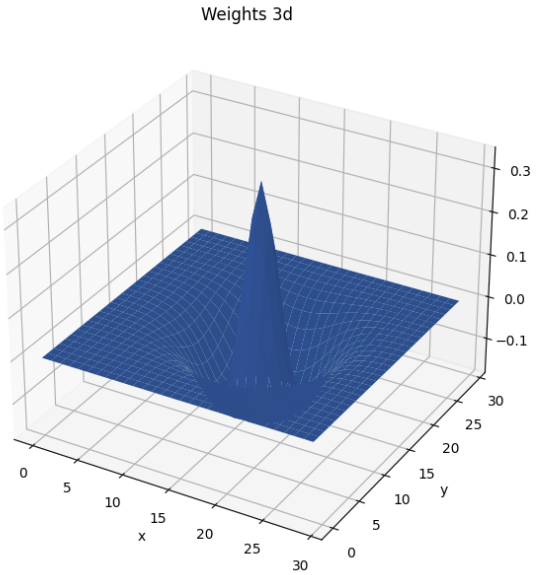


Figure 3.15: Weights for working memory 1d

3.4.4 Output

In this subsection, we explore the output in the trigger phase (figure 3.16) , post trigger phase (figure 3.17) and the deviation phase (figure 3.18).Each figure consists of 3 sub figures: left is the Input , middle is the output and the right is the template (upper bound in black , lower bound in orange), $u(x,t)$ in blue and $f(u(x,t))$ in red.

- **Trigger phase**: corresponding to the figure 3.16 , the loss is perfectly inside the lower bound upper bound template which is a really good result.
- **Post trigger phase**: We notice here (figure 3.17 and figure 3.19) that $f(u(x,t))$ is slightly under the lower bound which is normal since the input is very low but overall the result is very good regarding the output figure.
- **Deviation phase**: The deviated bump in the figure 3.18 and the figure 3.20 has more loss than the other one and seems that didn't follow the deviation really well. So we conclude that the parameters learned to memorize the bump but not enough to follow the deviation.

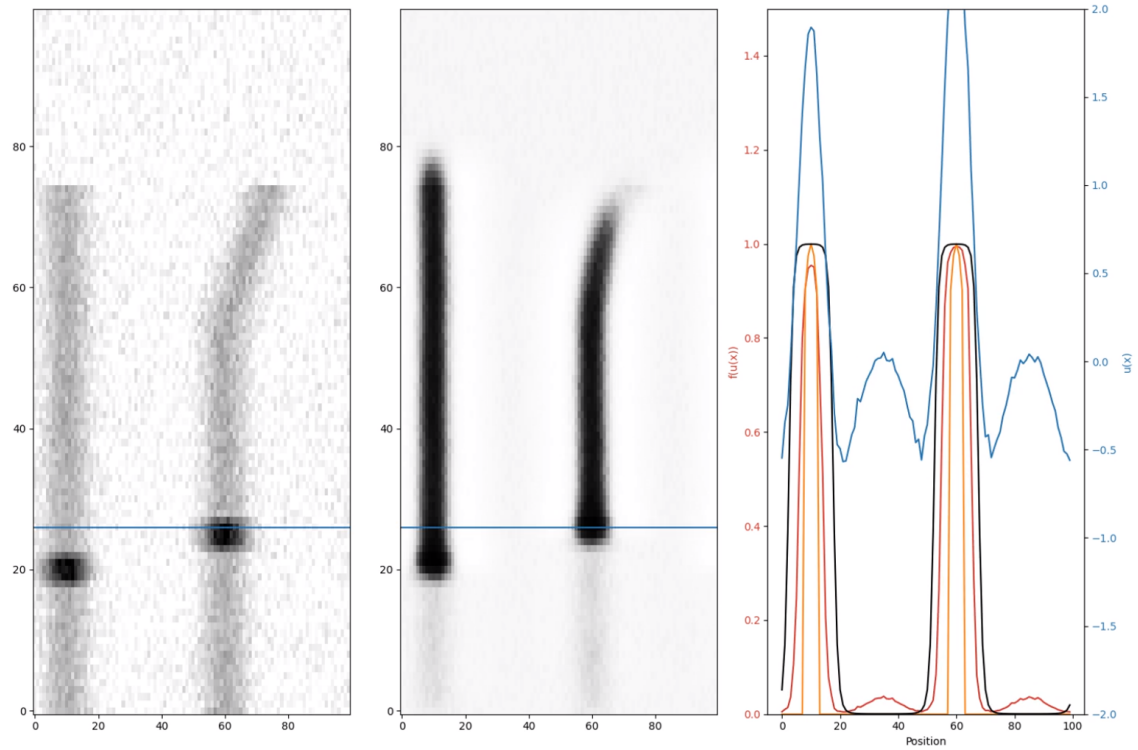


Figure 3.16: Slice of working memory in trigger phase

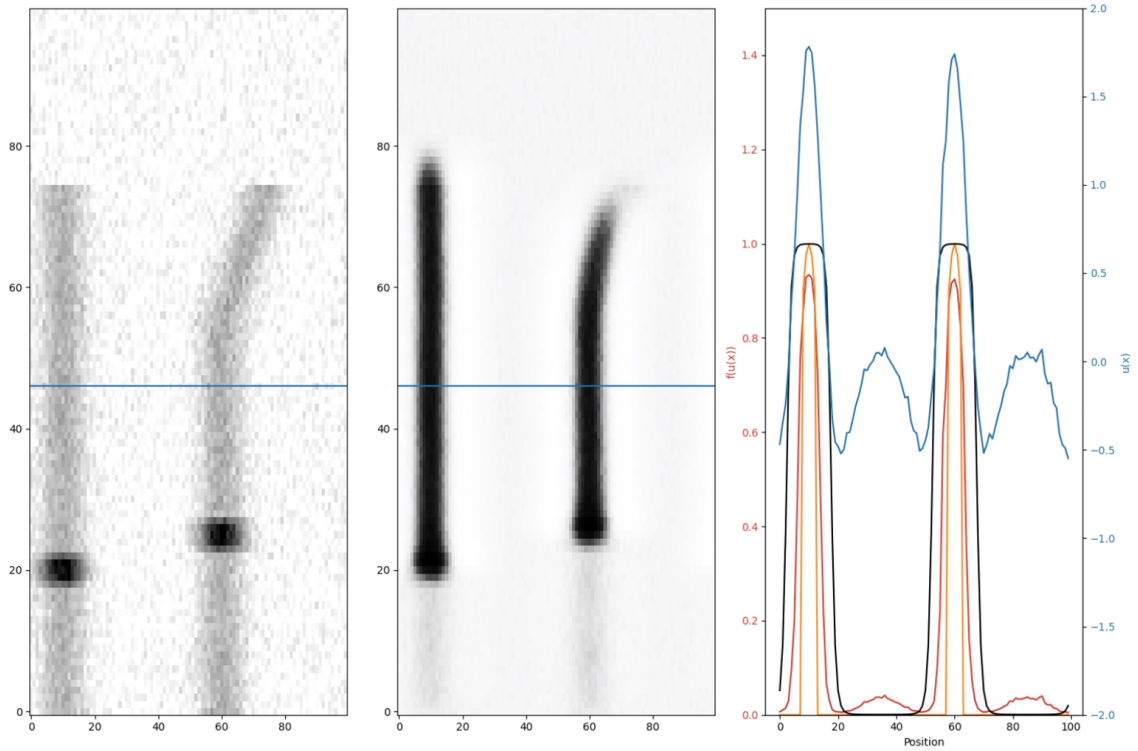


Figure 3.17: Slice of working memory in post trigger phase

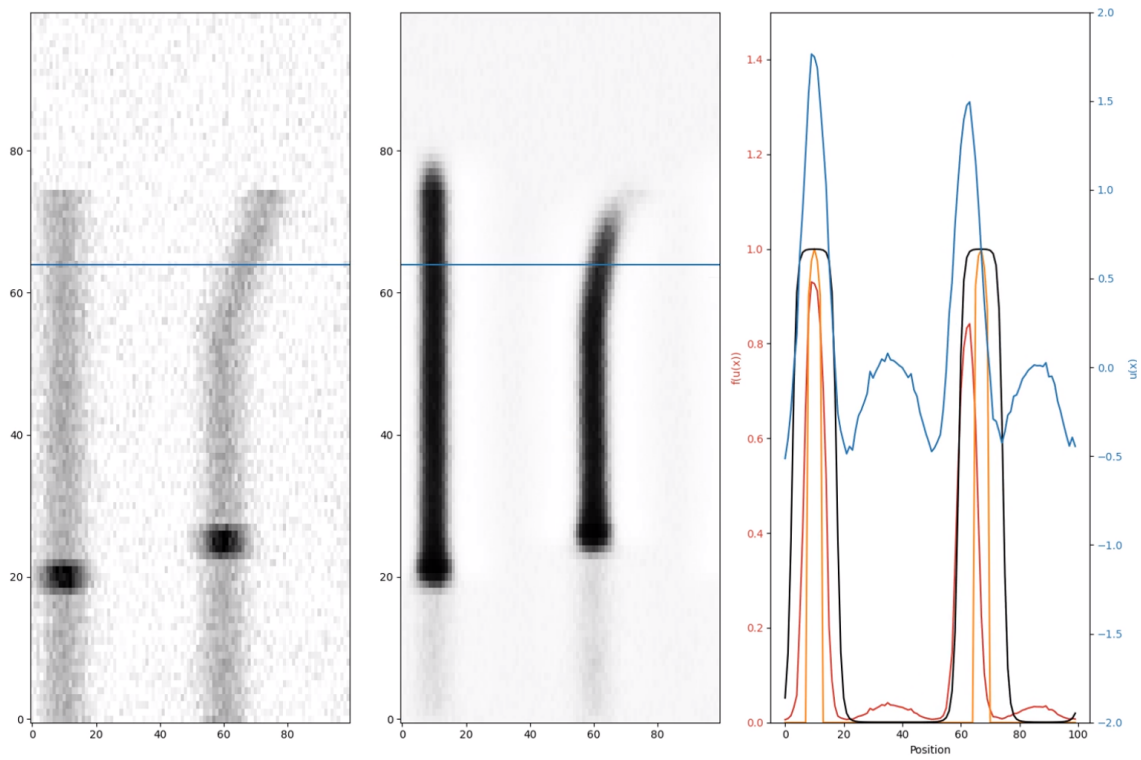


Figure 3.18: Slice of working memory in deviation phase

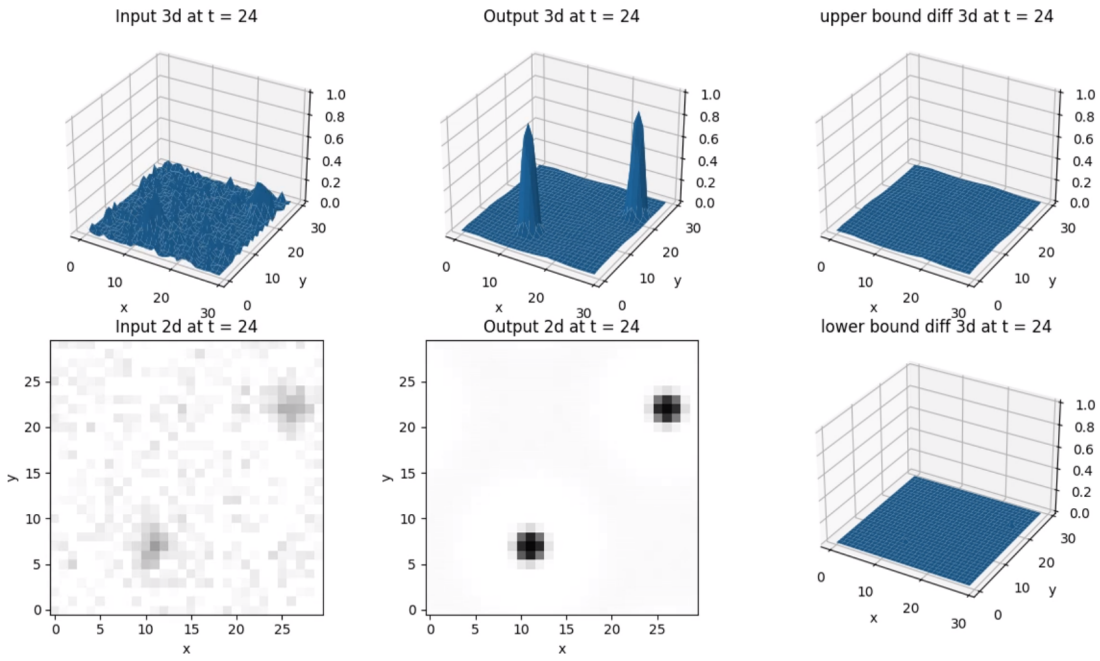


Figure 3.19: Slice of working memory 2d in post trigger phase

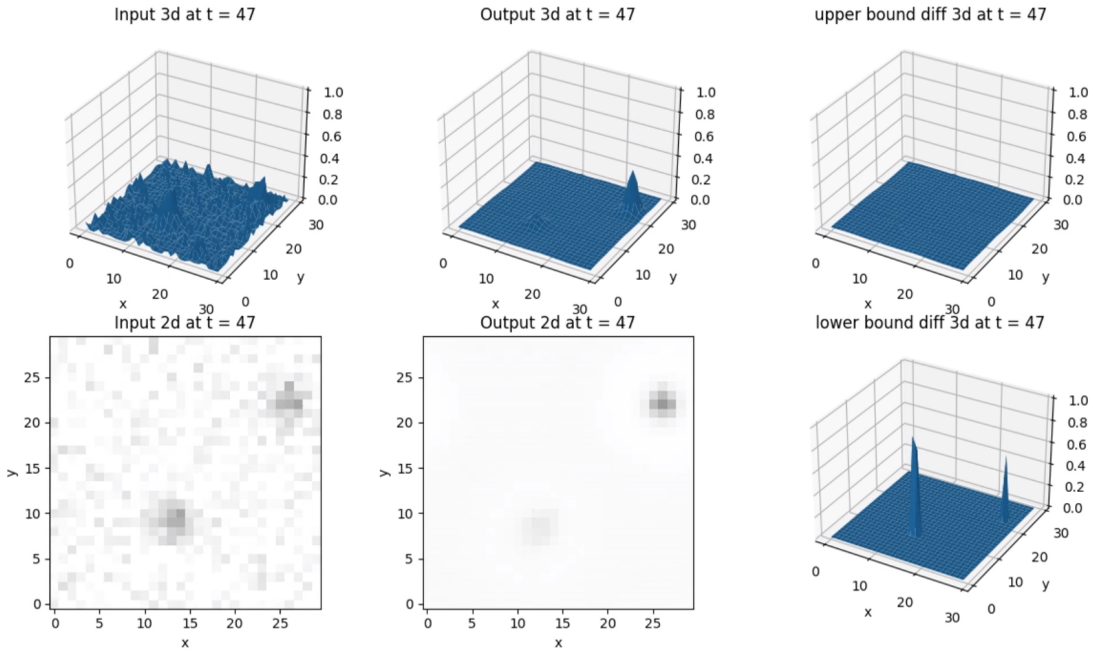


Figure 3.20: Slice of working memory 2d in deviation phase

Conclusion

In this internship we focused on implementing the dynamic neural field and optimizing it's parameters using only python and *pytorch* especially. We found some good results with competition scenario and decent results with the working memory scenario. In conclusion work proves that optimizing dynamic neural field parameters is possible using *pytorch*.

During the internship , I learned new things about this domain for the first time and I mastered using *pytorch* which is a powerful library for deep learning in python.

This work can be extended with the other transfer functions and weights functions as they can help with the hardware implementation and applied in the Dynamic formation of Self-Organizing Maps as explored in [6]

Bibliography

[1] R. L. Beurle. Properties of a mass of cells capable of regenerating pulses. Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 240(669):55–94, 1956.

[2] B. Chappet De Vangel and J. Fix. In the quest of efficient hardware implementations of dynamic neural fields: an experimentalstudy on the influence of the kernel shape. In Intemational Joint Conference on Neural Networks (IJCNN), 2016 Intemational Joint Conference on Neural Networks (IJCNN), Vancouver, Canada, July 2016.

[3] B. Chappet De Vangel, C. Torres-huitzil, and B. Girau. Randomly spiking dynamic neural fields. J. Emerg. Technol. Comput. Syst., 11(4), Apr. 2015.

[4] W. Erlhagen and E. Bicho. The dynamic neural field approach to cognitive robotics. Journal of Neural Engineering, 3(3):R36–R54, jun 2006.

[5] J. Fix. Template based black-box optimization of dynamic neural fields. Neural Networks, 46:40–49, 2013.

[6] J. Fix. Dynamic Formation of Self-Organizing Maps. In 10th International Workshop, WSOM, volume 295 of Advances in Intelligent Systems and Computing, pages 25–34, Mittweida, Germany, July 2014.