UNIVERSITÀ DI PISA

SYMBOLIC AND EVOLUTIONARY ARTIFICIAL INTELLIGENCE

< F9 Project >

Github repository (Public) **:** [DomenicoArmillotta/Symbolic_project-Hybrid_NA_DNN (github.com)]

Made by**:**

- **Domenico Armillotta -** 643020 - email: d.armillotta@studenti.unipi.it
- **Leonardo Bellizzi -** : 643019 - email: l.bellizzi@studenti.unipi.it

Contents

1. Mathematical explanation of the problem

The main objective is to create a Hybrid Neural Network composed of an input layer, a standard hidden layer and a final Non-Archimedean (NA) layer.

The problem that arises in networks like these, including the Full NA network, concerns the definition of the backpropagation function, which requires matrices inversions that are unstable in the continuous.

Since the main error is concentrated in the standard part of the loss function $L : \mathrm{E}^n \to \mathrm{E}$, the infinitesimal component risks creating instability during matrix inversion. This occurs because you are trying to backpropagate a finite quantity with infinitesimal component that may cancel each other out and cause instability.

To limit this instability, the focus is on the loss function that includes a standard part of the form $w_1 + w_2\eta + w_3\eta^2$, an infinitesimal gradient is obtained that, when backpropagated into the NA Layer, does not cause issues since only the infinitesimal part of the last layer is updated.

Howewer, the problem arises during the backpropagation of this same gradient into the standard part.

From an engineering perspective, a gradient of the following type is obtained:

$$\nabla \leftarrow \nabla \in o(\eta)$$

belonging to η (infinitesimal) but this gradient indicates a direction that is important to take as finite. For this reason, gradient will be multiplied by a finite quantity.

$$\nabla \leftarrow \nabla \in o(\eta) \cdot \alpha$$

This kind of operation allows to update the network without creating a significant instability.

However, an issue arises in passing the standard gradient in this kind of NN and here is where the KITE problem becomes useful.

KITE illustrates a situation where two points are separated by an optimal segment with respect to the first objective, and when approaching sufficiently with the inner product method, it is observed that the performed movement is not parallel to the second objective but as parallel as possible keeping the orthogonality with the first objective, that means without modifying the objective function.

The idea is therefore to project the gradient onto the space orthogonal to $w_1^{\perp}$ considering $w = w_1 + w_2\eta + w_3\eta^2$ where

$$P_{w_1^{\perp}}(\nabla) \to \nabla$$

$$w_1^{\perp} = \{z \in \mathbb{R}^n \mid z \cdot w_1 = 0\}$$

In this way, an orthogonal gradient to the finite component of w is obtained by projecting it onto the plane orthogonal to the previous gradient. This allows for a standard variation of the network that does not alter the finite part of the loss, as the finite part is not influenced by the variation of the update.

The new layer is thus defined as:

$$(l - \varphi\nabla) = l^T \cdot w - \varphi\nabla^T w$$

Where l rapresents the previous layer and φ is coefficient.

The gradient is chosen to be orthogonal to P, so $\varphi\nabla^T w = 0$ in the finite part. In this way, it is possible to train an Hybrid NN without numerical instability, as computations are performed only on the finite part, except for the infinitesimal update that is carried out forward without affecting the backpropagation.

It is important to note that the subsequent optimization of the third objective must not influence the optimization of the first and second objectives. Therefore, the gradient update is given by:

$$P_{w_1^{\perp} \cap w_2^{\perp}}(\nabla) \rightarrow \nabla$$

Eventually, the number of objectives should be smaller than the number of neurons in the standard last layer to have sufficient space for orthogonal manoeuvres.

2. Model components

2.1. - Input Data

The input data for the network consists of two regression problems, each associated with a different monosemion. Specifically, there are input data represented as X, along with corresponding outputs Y and Y2, which are related to the regression problems for monosemion 1 and monosemion 2, respectively.

More precisely, the input data consists of pairs of input values (X) and their corresponding output values (Y and Y2). These outputs are the targets aim to regress using our network. It is important to note that the regression problems associated with monosemion 1 and monosemion 2 are numerically distinct but exhibit similarities in their trends and patterns.

2.2. – Network architecture

A purposely simple architecture has been deliberately designed to address the challenges encountered more effectively. The designed architecture combines both Archimedean and non-Archimedean layers.

The overall structure of our network can be summarized as follows: It comprises an input neuron serving as the input layer, followed by 10 hidden neurons forming the hidden layer. Finally, the output neuron is represented by our custom Non-Archimedean neuron.

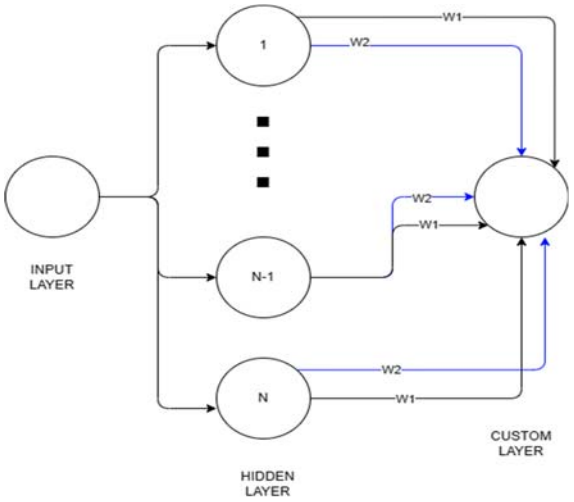


Figure 1: Network architecture

2.3 – Custom neuron

To formalize the problem effectively, a non-Archimedean layer was developed by redefining a custom neuron using PyThorc according to out specific requirements. This custom neuron serves as the building block of our network architecture.

The **custom neuron is equipped with weight1 and bias**, which are associated with **monosemion 1,** as well as **weight2 and bias2**, which are related to **monosemion 2**. These weights are appropriately frozen or unfrozen during the training process to ensure they are updated only when necessary and prevent incorrect updates in different phases.

Furthermore, the forward function of the neuron was redefined, to accommodate the training phase associated with monosemion 1. In this phase, only the weights and bias corresponding to monosemion 1 are taken into consideration during the forward pass, allowing us to optimize the regression for monosemion 1 separately.

```
class CustomNASingleNeuronLayer(nn.Module):
    def __init__(self, input_size, monosemio):
        super(CustomNASingleNeuronLayer, self).__init__()
        self.input_size = input_size
        self.monosemio = monosemio
        self.weight1 = nn.Parameter(torch.Tensor(input_size))
        self.bias = nn.Parameter(torch.Tensor(1))
        self.bias2 = nn.Parameter(torch.Tensor(1))
        self.weight2 = nn.Parameter(torch.Tensor(input_size))
        self.reset_parameters()

    def reset_parameters(self):
        torch.manual_seed(42)
        init.normal_(self.weight1)
        init.normal_(self.bias)
        init.normal_(self.bias2)
        init.normal_(self.weight2)

    def forward(self, x):
        if self.monosemio == 1:
            output = torch.matmul(x, self.weight1.t()) + self.bias
        elif self.monosemio == 2:
            output = torch.matmul(x, self.weight2.t()) + self.bias2
        else:
            raise ValueError("Invalid value for monosemio. Supported values are 1 and 2.")
        return output

    def set_monosemio(self, monosemio):
        self.monosemio = monosemio


    def freeze_parameters_w2(self):
        self.weight2.requires_grad = False
        self.bias2.requires_grad = False

    def unfreeze_parameters_w2(self):
        self.weight2.requires_grad = True
        self.bias2.requires_grad = True

    def freeze_parameters_w1(self):
        self.weight1.requires_grad = False
        self.bias.requires_grad = False

    def unfreeze_parameters_w1(self):
        self.weight1.requires_grad = True
        self.bias.requires_grad = True
```

Figure 2: Code snippet of custom neuron

Additional functionalities were implemented to freeze or unfreeze parameters associated with weight1 and weight2 separately, allowing fine-grained control over the training process for different monosemions.

These modifications allow to create a versatile and customizable non-Archimedean layer that effectively addresses the challenges of our regression tasks while facilitating efficient training and optimization of the network.

Additionally, another non-Archimedean layer was implemented in order to accommodate an hypothetical third infinitesimal. However, due to time constraints, it was not possible to fully extend the entire network to incorporate the management of this third monosemion.

2.4 – Autograd

One aspect of paramount importance to resolve the initial task was the modification of an internal component of PyThorc called Autograd.

Autograd is a fundamental module within the PyTorch deep learning library that enables automatic differentiation for gradient calculations in neural networks.

PyTorch simplifies gradient calculations by utilizing autograd. This module records tensor operations, constructs an acyclic computation graph, and automatically computes gradients using the chain rule. In practice, when tensor operations are performed in PyTorch, autograd tracks these operations, builds a graph to trace dependencies, and computes gradients accordingly.

In case study specific case, during the optimization phase of monosemion 1, it was essential to exclude the contribution of weights related to monosemion 2 from the gradient calculation. To achieve this, these weights were detached from the computation graph.

The same procedure was applied in the case of monosemion 2 as well. By detaching these weights, they were effectively excluded from the computation graph, as depicted in the following images.

This manipulation of the computation graph allows to selectively control the gradients and ensure that the weights associated with the undesired monosemions did not influence the optimization process. By detaching the weights, their impact was effectively drop on the network's weight updates, allowing s to focus only on the relevant weights and achieve the desired behaviour during training.

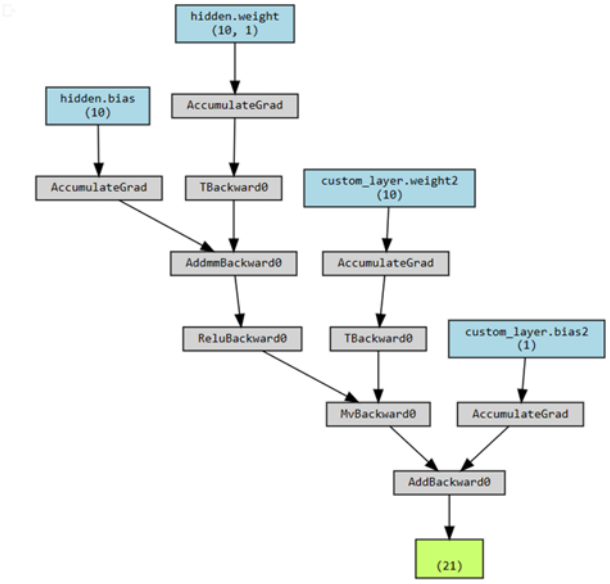


Figure 3: Autograd scheme for monosemio 2

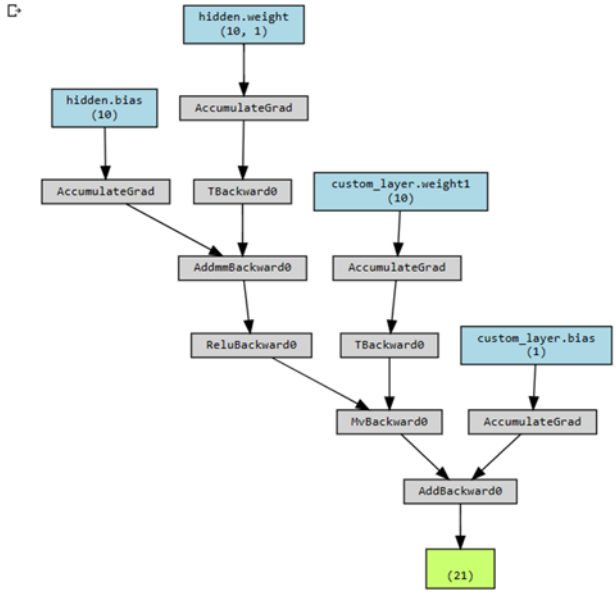


Figure 4: Autograd scheme for monosemio 1

2.5 – Update Hidden Gradient

One critical phase of the project involves updating the gradient of the hidden layer to improve the second objective while ensuring it does not compromise the significantly more important first objective. In this phase, a crucial step was done, known as the projection of the previous hidden gradient onto the vector orthogonal to the weight vector (Weight1) associated with the first objective.

By moving orthogonally to the first objective, the aim was to minimize the loss of the second objective (monosemion 2) while keeping the loss of the first objective (monosemion 1) unchanged, or at least minimizing its impact as much as possible.

To accomplish this, several tests and analyses were conducted on the intermediate values of tensors during training to verify the orthogonality and projection. These tests helped to ensure that the gradient updates of the hidden layer were performed in a manner that prioritized improving the second objective without negatively affecting the first objective. By projecting the previous hidden gradient onto the orthogonal vector, it is able to navigate the optimization process effectively, fine-tuning the network to achieve optimal performance for both objectives.

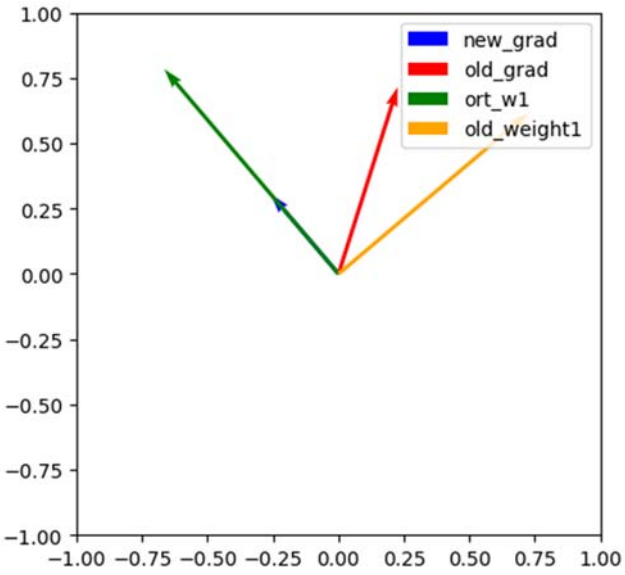


Figure 5: Orthogonality proof with projection result (new_grad) effectively orthogonal to w1

2.6 – Loss function

To calculate the loss in their model, they divided it based on the monosemion for which they wanted to evaluate the loss.

This division was necessary because, depending on the specific case, inhibition of the weights that were not relevant was needed. This was achieved relying on the custom forward function in the last non-Archimedean layer.

For instance, when calculating the loss for monosemion 1 using the input and target 1, evaluation of weights corresponding to monosemion 2 was exclueded. It is important to note that is necessary specify which loss is intended to calculate, whether it was loss1 or loss2.

During the training phase, the calculated losses played a crucial role in determining which monosemion to optimize at each step. By examining the losses, we could make informed decisions on which monosemion to prioritize and focus on optimizing.

This approach allowed us to adaptively optimize the network based on the specific monosemion and its corresponding objectives, enhancing the overall performance and achieving the desired results.

3. Training

The training phase is the most substantial component of the program, it involves a main loop that can be outlined using the following pseudocode:

Algorithm 1: Training

```
while step ≤ N do
    if lossM1 ≤ threshold then
        Update M2;
    else
        Update M1;
    end
end
```

Figure 6: Training pseudocode

A limit has been imposed on the number of epochs to control the duration of the training process.

The algorithm dynamically selects which weights to update for each input point, based on the loss. There are two phases in the algorithm: the forward phase and the backward phase.

When optimizing for monosemion 1, only the weights related to monosemion 1 (i.e., weight1 and bias of the custom layer, and the weights and biases of the hidden layer) are updated.

During the forward phase, only the weights of monosemion 1 are considered. In the backward phase, thanks to the modification of the autograd graph, the weights related to monosemion 2 are not considered when updating the hidden layer.

When optimizing for monosemion 2, the objective is to improve the second goal without degrading the first goal.

The forward phase proceeds as in a standard network, but the weights of monosemion 1 are not evaluated.

In the backward phase, the tensors related to monosemion 1 are eliminated from the autograd graph.

To update the hidden layer weights in a direction orthogonal to the weights of the first objective, the gradient is recalculated and projected onto the vector orthogonal to the weight vector w1. This ensures that monosemion 2 is improved without negatively impacting monosemion 1.

Additional variables have been introduced for printing graphs depicting the loss trend, gradient variation, and weight variation in the different layers. The intermediate steps were thoroughly analyzed, and as expected, the weights corresponding to the opposite monosemion did not change. The autograd functionality, projection technique, and the absence of loss increase in monosemion 1 after modifying the weights of monosemion 2 were verified through extensive testing.

These steps were particularly delicate, this phase was clearly the most challenging one of the whole work, as documented in the issue section. However, through careful testing and analysis all the issues were resolved.

4. Result

The obtained results align with all the previously discussed considerations. Both objectives demonstrate successful regression, as evident from the accompanying images.

The achieved outcomes are consistent with the theoretical expectations and validate the effectiveness of our approach. The regression performance for both objectives is satisfactory, as depicted in the visual representations.

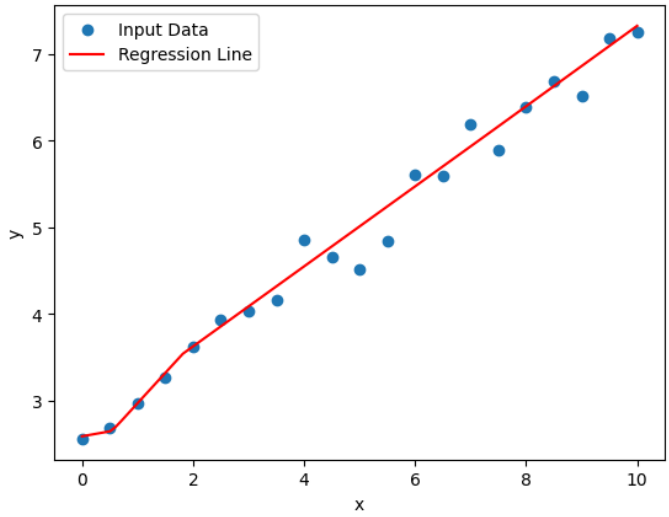


Figure 7: Output regression for Monosemio 1

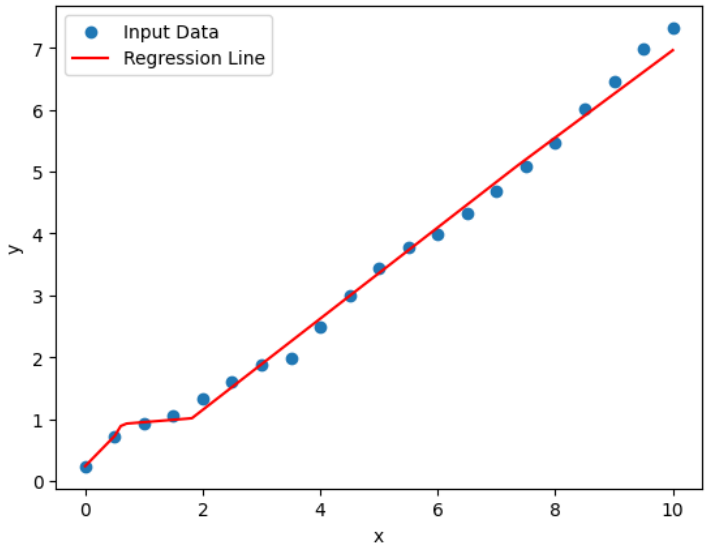


Figure 8: Output regression for Monosemio 2

The loss values exhibit a consistent and relatively rapid decrease, demonstrating the effectiveness of our approach. Additionally, the performance of the first monosemio aligns closely with that of the standard network, affirming the correctness of the implementation.

The observed reduction in loss values further supports the successful regression achieved by our model. This indicates that the network's optimization process effectively improves the accuracy of both monosemia, leading to more accurate predictions overall.

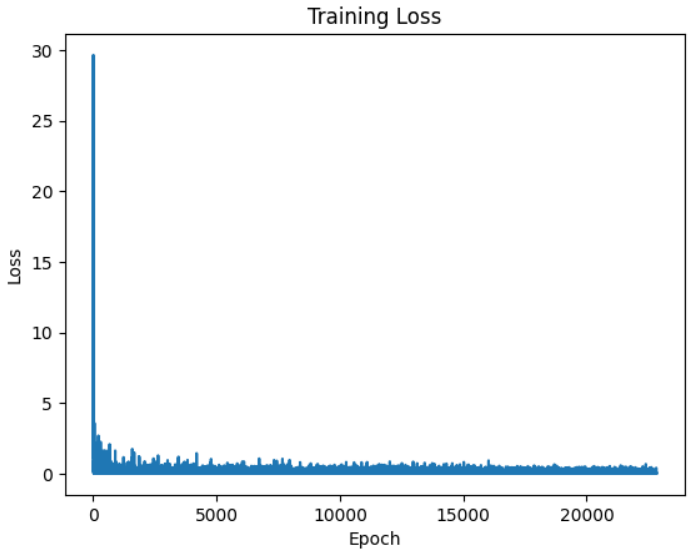


Figure 9: Loss for Monosemio 1 optimization

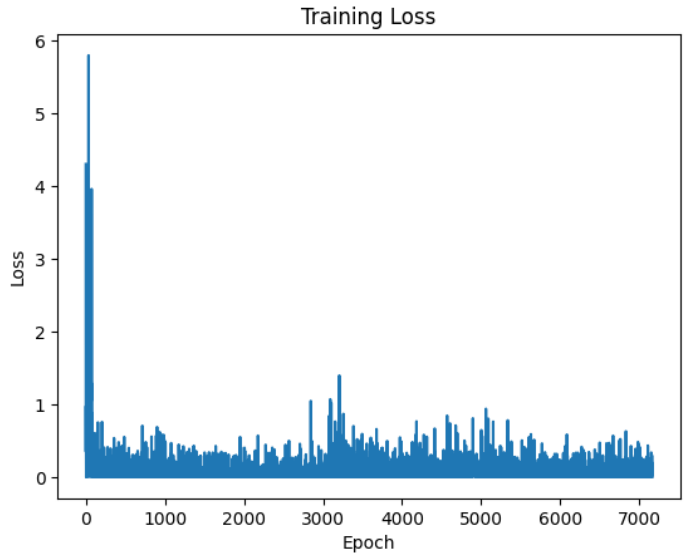
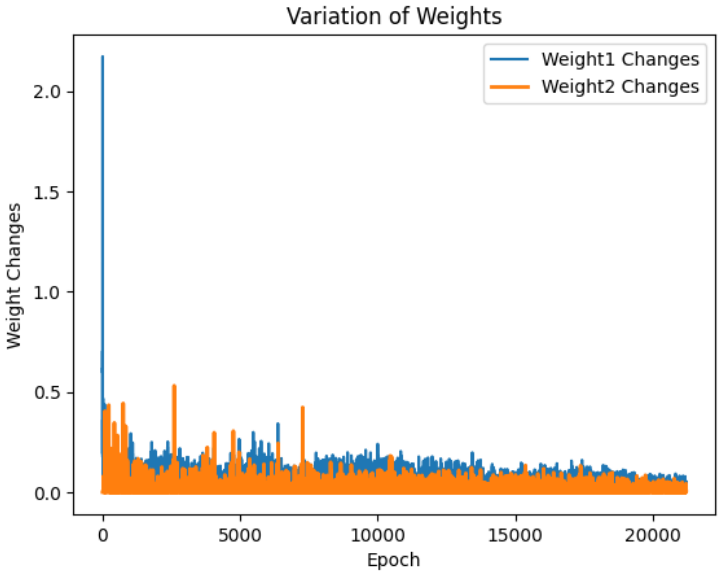
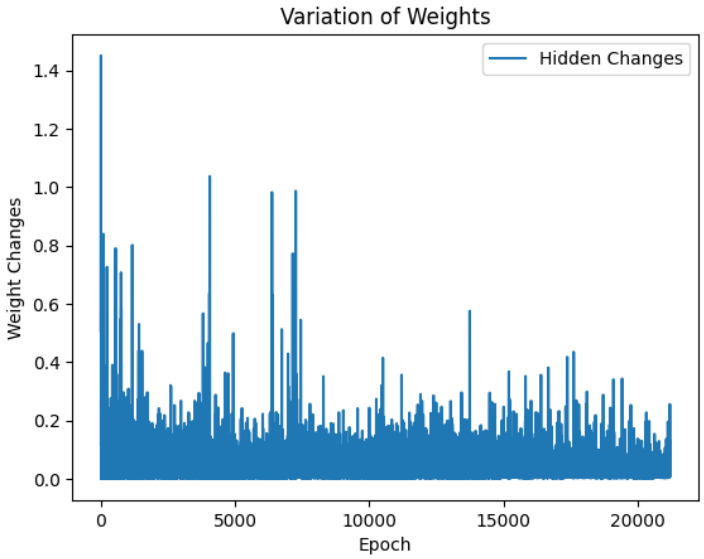


Figure 10: Loss for Monosemio 2 optimization





Figures 11-12: Weights changes for layers during steps

5. Test

To address the encountered challenges during the different implementations, several training tests were conducted, meticulously analyzing tensors and intermediate results. Tests were designed to ensure the correctness and effectiveness of modified approaches.

The key aspects during these tests were:

1. Attack/Detach from PyTorch's autograd computation graph of monoseed weights:

Behaviour of the autograd computation graph was carefully analyzed when manipulating the weights associated with monoseeds. This allowed to assess the impact of detaching or attacking these weights from the graph, ensuring that the desired optimizations were achieved.

2. Verification of the correct modification of weights based on the optimization phase of monosemion 1 or monosemion 2:

It was rigorously verified whether the weights were correctly adjusted during the optimization phase, depending on whether we were targeting monosemion 1 or monosemion 2. This verification process involved meticulous analysis of the weight updates to ensure the desired modifications were made accurately.

3. Orthogonality and projection of the gradient on the orthogonal vector to weight1:

Crucial point of this analysis was the orthogonality of the gradient with respect to weight1 and successfully projected it onto the orthogonal vector. This step was crucial to ensure the correct direction of the gradient and optimize the training process effectively.

4. Variation of the loss related to the monoseeds in the progress of training, considering the projection:
Throughout the training process, the variation of the loss associated with the monoseeds was monitored. Impact of the projection and assessed how it influenced the loss values was considered, allowing to evaluate the effectiveness of our approach.

By conducting these meticulous training tests, the correctness and efficacy of our implementations has been validated.

6. Issues

6.1 Incorrect regression

One of the initial challenges encountered was achieving regression on the first monosemion that matched the standard network. In order to address this, the last layer of the network was redfined to be Non-archimedean. Loss computation was also redfined, specifically considering only the weights related to monosemion 1.

After some analysis, it was discovered that the loss calculation was incorrect due to an additional dimension in the input passed to the function.

To test this, same data as input to both networks were provided. It was ensured that all weights were initialized equally for both the standard network (hidden weight, hidden bias, output weight, and output bias) and the custom network (hidden weight, hidden bias, monosemion 1 weight, monosemion 1 bias). It was examined all the intermediate results of the forward and backward processes, which allowed to identify the problem accurately.

6.2 Autograd correctness test

It was faced an additional challenge in determining the correct approach to exclude certain gradients during the backward pass based on the monosemion to optimize.

Specifically, while optimizing monosemion 1, it was needed to ensure that the weights associated with monosemion 2 stayed unchanged and they were not considered in the gradient calculation for updating the hidden layer.

To accomplish this, the custom layer class of the non-archimedean network to disable autograd was implemented. This function was designed to test the behaviour of the network during training using an ad hoc computation graph.

During the analysis of the computation graphs, it was observed two distinct scenarios.

In the first graph, we noticed the absence of the weights related to monosemion 1, specifically weight1 and bias.

Conversely, in the second graph, it was observed observed the absence of the weights associated with monosemion 2, namely weight2 and bias2. These observations provided further confirmation that the weights were indeed not changing for monosemions 1 and 2, respectively.

This detailed examination of the tensors related to the weights of monosemion 1 and monosemion 2 served to solidify our understanding that the intended exclusion of certain weights from the gradient calculations was successfully implemented.

```
w1 in m1: 365
tensor([ 0.3367, -0.0218, -0.8236, -0.5682, -1.5792, -0.2128,  2.1095, -0.6380,
        -0.4610, -0.8298])
w1 in m2: 366
tensor([ 0.3367, -0.0218, -0.8236, -0.5682, -1.5792, -0.2128,  2.1095, -0.6380,
        -0.4610, -0.8298])
```

Figure 12: Unchanged w1 values in both monosemio

6.3 Orthogonal and projected vector

A challenge related to tensor manipulation to execute the required operations for correctly updating monosemion 2 was also faced. Specifically, it was needed to project the gradient of the hidden layer onto the vector that is orthogonal to the weight vector of monosemion 1.

The aspect that presented difficulties was ensuring that the tensors had matching dimensions and shapes to enable the necessary computations. Challenges in aligning the dimensions and shapes of the tensors were encountered, which were crucial for successfully performing the operations.

Through careful manipulation and adjustments of the tensors, it was able to overcome these obstacles and ensure that the dimensions and shapes were aligned appropriately.

This allowed to accurately summarize and execute the required computations, resulting in the desired projection of the hidden layer gradient onto the orthogonal vector to the weight vector of monosemion 1.

Overall, by effectively addressing the tensor manipulation challenges, we achieved the necessary operations for updating monosemion 2 and successfully accomplished the desired objectives of the project.

6.4 Wrong implementation

Initially, implementation followed a two-step approach, with the first step focusing on the regression of monosemion 1, followed by a separate second step dedicated to the regression of monosemion 2. However, going further on implementation, it was significantly changed.

It was decided to merge these two steps into a single process. In this revised approach, for each data point, it was evaluated whether the loss of monosemion 1 falls below a predetermined threshold. If it does, we optimize the second target (monosemion 2). Conversely, if the loss of monosemion 1 exceeds the threshold, we prioritize the optimization of the first target (monosemion 1).

This change in the implementation approach needed several cascading modifications throughout the system. One of the major changes involved redefining the computation of the loss to accommodate this merged approach. It was necessary to carefully reconsider how the loss was calculated and adapt it to the updated workflow.

These alterations brought about a series of adjustments to various components and processes.

As a result, it was needed to revisit and revise several interconnected aspects to ensure the smooth integration of the new merged approach.

Eventually, this modification allowed to streamline the implementation, optimize the training process, and achieve better results in regression for both monosemion 1 and monosemion 2.

7. Considerations

A notable observation that was made pertains to the decrease in loss after updating the hidden layer during the phase dedicated to monosemium 2. Initially, it was an unexpected result. However, thorough further anlysis of the intermediate tensors during training and the projection operations, it was hypothesized that this phenomenon may be attributed to the high similarity between the regression lines of the two targets.

It is plausible that the modification of the hidden layer caused by optimizing monosemium 2 also had a positive impact on the loss of monosemium 1. Throughout our extensive analysis of the data, it was consistently observed a decrease in the loss values and did not encounter any instances where the loss increased.

This unexpected observation highlights an intriguing interplay between the two objectives, suggesting that improvements made in one monosemium can indirectly benefit the other, given their closely aligned regression lines. Further investigation and analysis would be necessary to fully comprehend the underlying dynamics at play.

```
l1 in m1: 17 sampple tensor([6.])
tensor(0.0268, grad_fn=<MseLossBackward0>)
l2 in m1: 17
tensor(0.0143, grad_fn=<MseLossBackward0>)
l1 in m2: 5 sampple tensor([6.])
tensor(0.0033, grad_fn=<MseLossBackward0>)
l2 in m2: 5
tensor(0.0143, grad_fn=<MseLossBackward0>)
```

Figure 13: Evidence of the slight improvement of M2 on M1

8. Future improvements

Several potential avenues for further analysis and extensions have emerged from this work. Unfortunately, due to time limitations, it was not possible to implement them. However, this idea could be crucial in a further exploration of the same case study:

1. Extension to the third monosemy: The addition of a third objective would require updating the corresponding gradient while ensuring it does not compromise the optimization of the first two objectives.

2. . Complexity of regression functions: Introducing more intricate functions to be regressed upon would challenge the network's capacity to learn and generalize effectively.

3. . Altering the nature of the problem: Experimenting with a network featuring additional hidden layers would introduce greater complexity during the backward phase and potentially yield more intricate optimization dynamics.

4. . Exploring a fully non-Archimedean network: Investigating the possibility of constructing an entirely non-Archimedean network could provide valuable insights into the benefits and challenges associated with this alternative approach.

These ideas represent opportunities for future research, building upon the foundations established in this project.

9. References

Custom layer : Writing a Custom Layer in PyTorch | by Auro Tripathy | Medium

Autograd : Autograd mechanics — PyTorch 2.0 documentation

torch.Tensor.requires_grad_ — PyTorch 2.0 documentation

Visualize Autograd : szagoruyko/pytorchviz: A small package to create visualizations of PyTorch execution graphs (github.com)