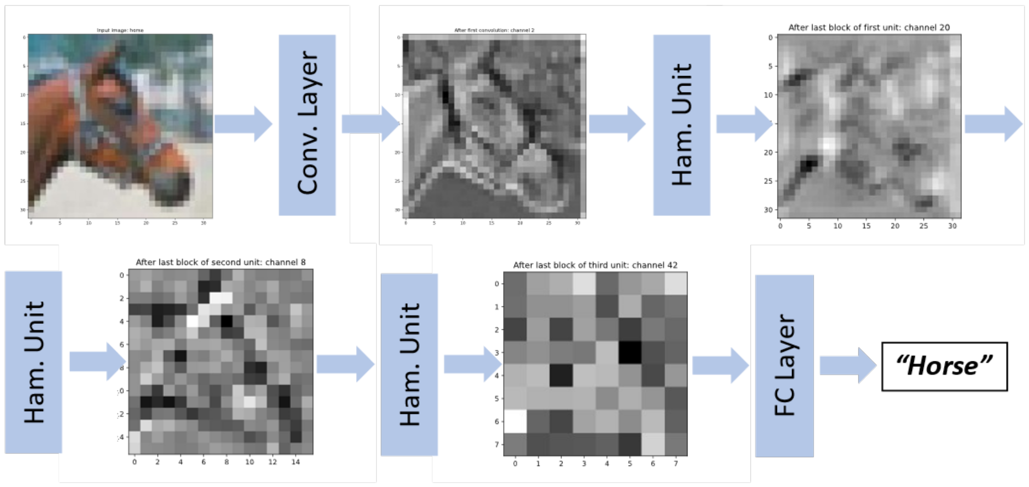
Automatic Control Laboratory

Image classification nets using Hamiltonian neural networks

Semester Project Report

Student: Ivan-Daniel Sievering
Supervisor: Clara Galimberti
Professor: Giancarlo Ferrari Trecate


Input image: horse
Conv. Layer
After first convolution: channel 2
Ham. Unit
After last block of first unit: channel 20
Ham. Unit
After last block of second unit: channel 8
Ham. Unit
After last block of third unit: channel 42
FC Layer
"Horse"

École Polytechnique Fédérale de Lausanne

June 14, 2021

Contents

1. Introduction

The use of Deep Learning has exploded in recent years. More, the number of fields in which it is applied is getting broader and broader. Despite numerous enhancements, two main issues remain: the vanishing gradient and the exploding gradient.

In their paper, Haber and Ruthotto [1] propose a new architecture that avoids these problems: the Hamiltonian Neural Network. The central concept behind it is to consider the propagation as the discretisation of a system of ordinary differential equations. From this definition, it is possible to analyse the stability of the system and to design it to never explode or vanish.
Based on this finding, various Hamiltonian Networks have been proposed ([1],[2] & [3]). Galimberti et al. [3] have proposed a unified notation to encompass all of them in a single equation and have introduced two new ones.
One of the fields where Deep Learning has the greatest success is Image Classification. A complete structure to use Hamiltonian Networks for this task has been proposed by Chang et al. [2] and shows promising results.

The first goal of this project is to train Hamiltonian Neural Networks for the Image Classification task. The framework has been implemented successfully and offers various options. More, it can be used with various type of Hamiltonian Networks.
The second goal was to assess the performance of the different existing Hamiltonian Networks. The Two-Layers Hamiltonian from Chang, Meng et al. [2] has been improved and shows excellent performance. Even more excellent performance has been achieved with the J1 Hamiltonian Network, which was introduced by Galimberti et al. [3]. These results are comparable to standard networks but do not overperform the best state-of-the-art networks.

The report is organised as follows. First, the main theory about Hamiltonian Neural Networks and their application to Image Classification is introduced. Second, the framework that has been implemented is presented. Then, the results obtained using the different Hamiltonians are shown. Afterwards, their performance is discussed and compared to standard and state-of-the-art networks. Finally, the report is concluded.

2. Theory and Related Works

In this chapter, the main Hamiltonian Deep Neural Networks (H-DNN) theory is introduced. Then, different Hamiltonians Networks are presented. Finally, a network that applies H-DNN to the Image Classification task is described. A deeper and more complete description of H-DNN, as well as the mathematical proofs, can be found in the related academic papers ([1],[2] & [3]).

2.1 Neural Networks as dynamic system

H-DNN are a specific architecture of Neural Network (NN) introduced by Haber and Ruthotto [1]. In the following, the main steps of the former paper are presented.
First, defining the vector of neurons' activation at depth j as $\boldsymbol{y_j}$, a transition between two layers of a NN can be defined by the following equation:

$$\boldsymbol{y_{j+1}} = f(\boldsymbol{y_j}) \tag{2.1}$$

In the case of Residual Networks, the equation 2.1 can be rewritten as:

$$\boldsymbol{y_{j+1}} = \boldsymbol{y_j} + \sigma(\boldsymbol{K_j}\boldsymbol{y_j} + \boldsymbol{b_j}) \tag{2.2}$$

Where σ is the activation function (element-wise), typically a hyperbolic tangent (Tanh) or a Rectified Linear Unit (ReLU). $\boldsymbol{K_j}$ is the weight matrix at layer j and $\boldsymbol{b_j}$ the bias vector at the same layer. $K \in \mathbb{R}^{N\times m\times n}$, where N is the number of layers, m the number of neurons in the previous layer and n the number of layer in the current layer. Note that for convolutional layers, $\boldsymbol{K_j}$ is a convolutional operator without any influence on the following.
Let us add a multiplicative factor, h (>0), ahead of the activation function, leading to:

$$\boldsymbol{y_{j+1}} = \boldsymbol{y_j} + h\sigma(\boldsymbol{K_j}\boldsymbol{y_j} + \boldsymbol{b_j}) \tag{2.3}$$

Thanks to this modification, equation 2.3 can be considered as the Euler discretization of a dynamic system which would be defined by the following Ordinary Differential Equation (ODE):

$$\dot{\boldsymbol{y}}(t) = \sigma(\boldsymbol{K}^T(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)) \tag{2.4}$$

From this definition as a dynamic system over time t, its stability can be studied. For Deep Learning (DL), both the direct and inverse problem (propagation and learning) have to be well-posed.
The forward stability of a system can be assessed if the ODE meets some conditions, i.e.:

1. h is sufficiently small;
2. $\boldsymbol{K}$ and $\boldsymbol{b}$ change sufficiently smoothly;
3. the i-th eigenvalue (λ_i) of the Jacobian ($\boldsymbol{J}$) is negative or equal to zero for all i.

For the first condition, h can be tuned for each problem to meet Lemma 1 from Haber and Ruthotto [1]. For the second condition, the smoothness of the variation of K can be achieved by regularisation. The paper proposes to add to the regular loss two terms:

$$R(\boldsymbol{K}) = \frac{1}{2h}\sum \|\boldsymbol{K_j} - \boldsymbol{K_{j-1}}\|_F^2 \text{ and } R(\boldsymbol{b}) = \frac{1}{2h}\sum (\boldsymbol{b_j} - \boldsymbol{b_{j-1}})^2 \tag{2.5}$$

F refers to the Frobenius norm. The third condition corresponds to:

$$\max_{i=1,2,\ldots,n} Re(\lambda_i(\boldsymbol{J}(t)) \leq 0, \forall t \in [0,T] \tag{2.6}$$

For the "ResNet" example, it is:

$$\max_{i=1,2,\ldots,n} Re(\lambda_i(\boldsymbol{K}(t)) \leq 0, \forall t \in [0,T] \tag{2.7}$$

These three conditions ensure that the forward propagation of the system is stable. However, eigenvalues with very negative values lead to systems were all the points converge to one (or various) fixed point. This is not an interesting behaviour because the difference between features would be lost. In such case, the inverse problem (learning) is ill-posed. Thus, the inequality 2.7 can be changed to:

$$\max_{i=1,2,\ldots,n} Re(\lambda_i(\boldsymbol{J}(t)) \approx 0, \forall t \in [0,T] \tag{2.8}$$

The previous equation, along with the small h value and the smooth change in $\boldsymbol{K}$ and $\boldsymbol{b}$, ensure that both the direct and inverse problem (forward and learning) are stable, and so that neither the gradient explodes nor vanishes. Thus, by verifying the conditions mentioned above, one can ensure (by restraining the space of solutions) that these two critical problems will not appear. Consequently, these NN can be arbitrarily deep, which is a great feature as deepness tend to increase the performance of the networks.

2.2 Hamiltonians Networks

2.2.1 Symmetric Hamiltonian Network

One way to enforce the conditions from last section is by recasting the forward propagation as a Hamiltonian system and to study its stability. This type of system is interesting because it conserves the energy. A general structure for the Hamiltonian system is:

$$\begin{aligned} \dot{\boldsymbol{y}}(t) &= -\nabla_{\boldsymbol{z}} \boldsymbol{H}(\boldsymbol{y}, \boldsymbol{z}, t) \\ \dot{\boldsymbol{z}}(t) &= -\nabla_{\boldsymbol{y}} \boldsymbol{H}(\boldsymbol{y}, \boldsymbol{z}, t) \end{aligned} \tag{2.9}$$

H is the Hamiltonian function ($\mathbb{R}^N \times \mathbb{R}^N \times [0,T] \to \mathbb{R}$), which can have the following shape when the energy is conserved (no time dependence):

$$\boldsymbol{H}(\boldsymbol{y}, \boldsymbol{z}) = \frac{1}{2}\boldsymbol{z}^T\boldsymbol{z} + f(\boldsymbol{y}) \tag{2.10}$$

Merging the equations leads to:

$$\ddot{\boldsymbol{y}}(t) = \nabla_{\boldsymbol{y}} f(\boldsymbol{y}(t)) \tag{2.11}$$

Haber and Ruthotto [1] proposed, inspired by ResNet (equation 2.4), to rewrite equation 2.11 as:

$$\ddot{\boldsymbol{y}}(t) = \sigma(\boldsymbol{K}(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)) \tag{2.12}$$

As before, to ensure no vanishing or exploding gradient: real part of the eigenvalues of K have to be near to zero. They propose to redefine the ODE in a symmetric way:

$$\begin{aligned} \dot{\boldsymbol{y}}(t) &= \sigma(\boldsymbol{K}(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)) \\ \dot{\boldsymbol{z}}(t) &= -\sigma(\boldsymbol{K}^T(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)) \end{aligned} \tag{2.13}$$

With this definition, the network is intrinsically stable, whenever the two other conditions are met (h sufficiently small and smooth variation of K and b).
The discretisation of the previous system (using leapfrog discretisation and Verlet integration) results in:

$$\begin{aligned} \boldsymbol{z}_{j+\frac{1}{2}} &= \boldsymbol{z}_{j-\frac{1}{2}} - h\sigma(\boldsymbol{K}_j^T\boldsymbol{y}_j + \boldsymbol{b}_j) \\ \boldsymbol{y}_{j+1} &= \boldsymbol{y}_j - h\sigma(\boldsymbol{K}_{j+\frac{1}{2}}^T\boldsymbol{y_j} + \boldsymbol{b_j}) \end{aligned} \tag{2.14}$$

2.2.2 Two-Layers Hamiltonian

In the Chang et al. paper [2] the Two-Layer Hamiltonian is proposed, it is an extension of the previous network. The main difference with the original Hamiltonian Network is that this network has two layers. This is motivated by the fact that a two-layer neural network can approximate any function (that respects some conditions). The following ODE are proposed (similar to equation 2.13):

$$\begin{aligned} \dot{\boldsymbol{y}} &= \boldsymbol{K_1}^T(t)\sigma(\boldsymbol{K_1}(t)\boldsymbol{z}(t) + \boldsymbol{b_1}(t)) \\ \dot{\boldsymbol{z}} &= -\boldsymbol{K_2}^T(t)\sigma(\boldsymbol{K_2}(t)\boldsymbol{y}(t) + \boldsymbol{b_2}(t)) \end{aligned} \tag{2.15}$$

Where y and z are separated using equal channel-wise partition and then simply reconcatenated. The discretisation of the network is:

$$\begin{aligned} \boldsymbol{y}_{j+1} &= \boldsymbol{y}_j + h\boldsymbol{K}_{j1}^T\sigma(\boldsymbol{K}_{j1}\boldsymbol{z}_j + \boldsymbol{b}_{j1}), \\ \boldsymbol{z}_{j+1} &= \boldsymbol{z}_j + h\boldsymbol{K}_{j2}^T\sigma(\boldsymbol{K}_{j2}\boldsymbol{y}_j + \boldsymbol{b}_{j2}) \end{aligned} \tag{2.16}$$

Like the previous one, this network is stable (forwardly and backwardly) if the two other conditions are met. For the smoothness decay, they propose to use (similarly to equation 2.5):

$$R(\boldsymbol{K}) = h\sum_{j=1}^{T-1}\sum_{k=1}^{2}\left\|\frac{\boldsymbol{K}_{j,k} - \boldsymbol{K}_{j+1,k}}{h}\right\|_F^2 \tag{2.17}$$

2.2.3 J1 and J2 Hamiltonian

The paper from Galimberti et al. proposes a unified framework that includes both the Hamiltonian Networks and the Anti-Symmetric Networks. The latter is another way to enforce the three conditions that lead to bounded gradients.
To do so, the following ODE, which defines a time-varying Hamiltonian system, is used:

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{J}(\boldsymbol{y}, t)\frac{\partial \boldsymbol{H}(\boldsymbol{y}, t)}{\partial \boldsymbol{y}} \tag{2.18}$$

J is a matrix that has to be skew-symmetric and H is the Hamiltonian function. The following Hamiltonian is proposed:

$$\boldsymbol{H}(\boldsymbol{y}(t), t) = [log(cosh(\boldsymbol{K}(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)))]^T\boldsymbol{1} \tag{2.19}$$

log and *cosh* are applied element-wisely. **1** is a vector full of ones. Thus, equation 2.18 becomes:

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{J}(\boldsymbol{y}, t)\boldsymbol{K}^T(t)Tanh(\boldsymbol{K}(t)\boldsymbol{y}(t) + \boldsymbol{b}(t)) \tag{2.20}$$

Which corresponds to (when discretized with the forward Euler method) :

$$\boldsymbol{y}_{j+1} = \boldsymbol{y}_j + h\boldsymbol{J}\boldsymbol{K}_j^T Tanh(\boldsymbol{K}_j \boldsymbol{y}_j + \boldsymbol{b}_j) \tag{2.21}$$

They show that this notation (2.20) maintains all the properties of the already existing networks while offering new perspectives. The Symmetric Hamiltonian Network and the Two-Layers Hamiltonian Network can be rewritten using this notation. The shape of K and J for each one is presented in table 2.1.
Galimberti et al. [3] define two new Hamiltonian Networks: J1 and J2. These two networks do not have conditions on K and only restrain the shape of J, which lead to more parameters in the network and thus more learning potential. Their definition can be found in table 2.1. Note that in some parts of this report, J1 and J2 will be used with ReLU instead of Tanh.
Again, a similar smoothness regularisation like in equation 2.5 and 2.17 is added to the loss::

$$R(\boldsymbol{K}) = \frac{h}{2}\sum_{j=1}^{N-1} \|\boldsymbol{K}_j - \boldsymbol{K}_{j-1}\|_F^2 \text{ and } R(\boldsymbol{b}) = \frac{h}{2}\sum_{j=1}^{N-1} \|\boldsymbol{b}_j - \boldsymbol{b}_{j-1}\|^2 \tag{2.22}$$

Table 2.1: Different Hamiltonians definition based on the unified notation [3] (eq. 2.21 & 2.20)

Name	$\boldsymbol{K(t)}$	$\boldsymbol{J}(\boldsymbol{y}, t)$	Discretisation method
Symmetric [1]	$\begin{bmatrix} 0 & \boldsymbol{K}_0(t) \\ -\boldsymbol{K}_0(t) & 0 \end{bmatrix}$ is invertible	$\boldsymbol{J}(\boldsymbol{y}, t)\boldsymbol{K}^T(t) = \boldsymbol{I}$	Forward Euler
Two-Layers [2]	$\begin{bmatrix} 0 & \boldsymbol{K}_1(t) \\ \boldsymbol{K}_2(t) & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & \boldsymbol{I} \\ -\boldsymbol{I} & 0 \end{bmatrix}$	Verlet
J1 [3]	Free	$\begin{bmatrix} 0 & \boldsymbol{I} \\ -\boldsymbol{I} & 0 \end{bmatrix}$	Forward Euler
J2 [3]	Free	$\begin{bmatrix} 0 & 1 & ... & 1 \\ -1 & 0 & ... & 1 \\ ... & ... & ... & ... \\ -1 & -1 & ... & 0 \end{bmatrix}$	Forward Euler

2.3 Structure for Image Classification

One domain of application for H-DNN is Image Classification, i.e. learn to determine the label associated with an image. In the Chang et al. paper [2], a network for this aim is proposed. An illustrative representation coming from their paper is shown in figure 2.1.
The network uses the Two-Layers Hamiltonian presented in section 2.2. ReLU is used as an activation layer, and K is defined as a convolutional operator. A representation of this basic *block* is shown in figure 2.2.
Units are built by concatenating various of these *blocks* together and then adding a pooling and a padding layer. The pooling is achieved using average pooling on each channel. The zeros-padding is done by appending channels full of zeros. Note that the last unit does not have zero-padding because it is not interesting to add inconsistent information before the classification.
The entire network is then built by first defining a convolutional layer (in order to increase the number of channels of the input), then stacking a certain number of *units* and finally adding a fully connected layer that will classify the image. For more implementation details, please look directly at the paper [2].
This network shows competitive performance with state-of-the-art networks (ResNet and RevNet) on the STL-10, CIFAR-10 and CIFAR-100 datasets, as shown in figure 2.4. More, they show that they perform better than ResNet for fewer available data, which can be a struggle in many DL applications (figure 2.3). Lastly, the paper showed the efficiency of Hamiltonian Networks by training a 1202-layer network without vanishing or exploding gradient issues.

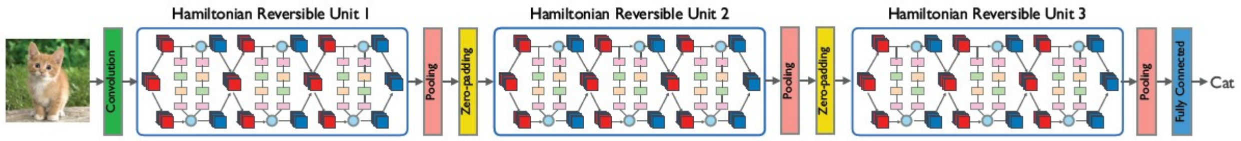

Hamiltonian Reversible Unit 1
Hamiltonian Reversible Unit 2
Hamiltonian Reversible Unit 3
Convolution
Pooling
Zero-padding
Pooling
Zero-padding
Pooling
Fully Connected
Cat

Figure 2.1: Illustrative representation of the Image Classification structure from Chang et al. paper [2].

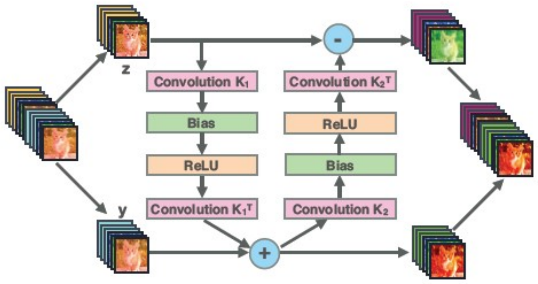


Figure 2.2: Illustrative representation of the Two-Layers Hamiltonian block from Chang et al. paper [2].

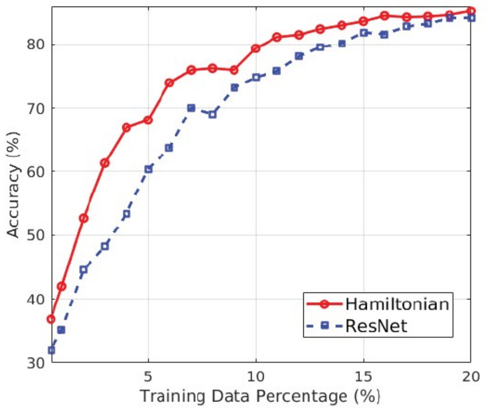
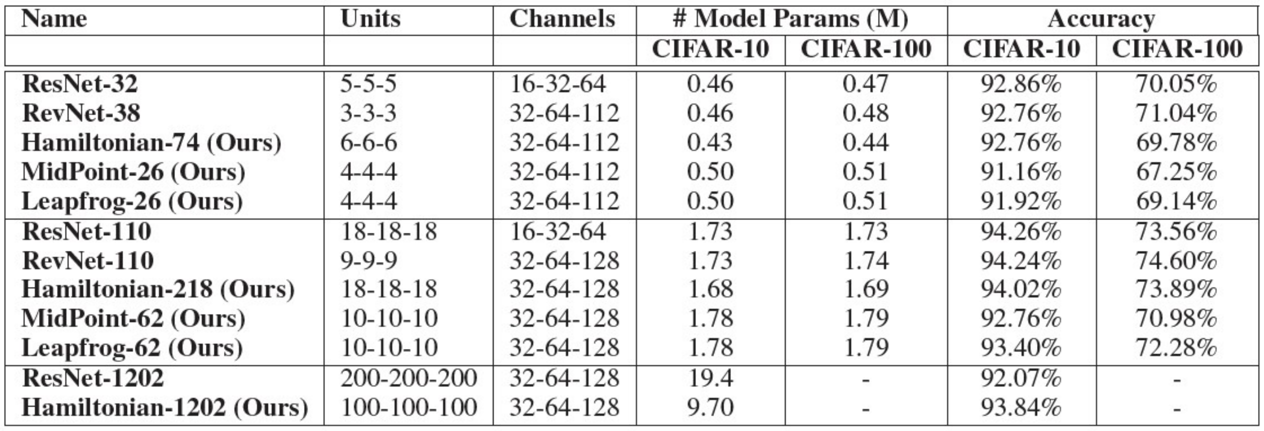


Figure 2.3: Performance of the Hamiltonian Chang et al. structure vs ResNet on reduced datasets. Figure from Chang et al. paper [2].



Name	Units	Channels	# Model Params (M)		Accuracy	
			CIFAR-10	CIFAR-100	CIFAR-10	CIFAR-100
ResNet-32	5-5-5	16-32-64	0.46	0.47	92.86%	70.05%
RevNet-38	3-3-3	32-64-112	0.46	0.48	92.76%	71.04%
Hamiltonian-74 (Ours)	6-6-6	32-64-112	0.43	0.44	92.76%	69.78%
MidPoint-26 (Ours)	4-4-4	32-64-112	0.50	0.51	91.16%	67.25%
Leapfrog-26 (Ours)	4-4-4	32-64-112	0.50	0.51	91.92%	69.14%
ResNet-110	18-18-18	16-32-64	1.73	1.73	94.26%	73.56%
RevNet-110	9-9-9	32-64-128	1.73	1.74	94.24%	74.60%
Hamiltonian-218 (Ours)	18-18-18	32-64-128	1.68	1.69	94.02%	73.89%
MidPoint-62 (Ours)	10-10-10	32-64-128	1.78	1.79	92.76%	70.98%
Leapfrog-62 (Ours)	10-10-10	32-64-128	1.78	1.79	93.40%	72.28%
ResNet-1202	200-200-200	32-64-128	19.4	-	92.07%	-
Hamiltonian-1202 (Ours)	100-100-100	32-64-128	9.70	-	93.84%	-

Figure 2.4: Results obtained by the Chang et al. [2], "Ours" refers here to their network. The H-DNN shows competitive performance with respect to ResNet and RevNet.

3. Framework

For this project, a framework to train H-DNN for the Image Classification task has been implemented. The framework uses a base architecture highly similar to what was proposed by Chang et al. [2]. It is separated into three main parts: the learning process (how the network is trained), the whole network structure and the Hamiltonian *blocks*. Each one of these is defined by a *dataclass*. In its main file (**run.py** in my case), the user can define these three structures, select the dataset to work with and then directly run the function *train_and_test* that will create the network, train it and then test it. In the appendices, a minimal code example is presented (appendix A.1.1) as well as the code organisation (appendix A.1.2)
In the next sections, the learning process (3.1), the network structure (3.2) and the Hamiltonian layer (3.3) will be presented in detail. For each one, the *dataclass* that defines the corresponding part is described.

3.1 Learning Process

This project's learning process (loss computation, weights update, ...) is very standard. The only novelty is the weights smoothness regularisation loss (its exact implementation is presented in Appendix A.2.1). The main steps of the learning process are described in the pseudo-code 1. Please pay attention to the variables used in this code as they are the ones from the *LearningParameters* structure. All the parameters defining the learning process are briefly described in table 3.1. For the ones with an asterisk, more details are provided in appendix A.2.1.

Table 3.1: Explanation of *LearningParameters* parameters. Together, they define the whole training process. Parameters noted with an asterisk are further explained in appendix A.2.1.

Parameters	Description	Data type
training_steps_max	Total number of training steps (i.e. network udpates)	int
batch_size	Size of a batch	int
lr	Initial learning rate (lr) for the optimizer	float
lr_decay_at	List of epochs indicating when a lr decay has to happen	List
lr_decay	Value of the lr decay ($lr_{new} = lr_{old} * lr\_decay$)	float
wd*	Weight decay regularisation weight	float
alpha*	Weight smoothness regularisation weight	float
b_in_reg*	Add (or not) the weight smoothness regularisation of b	Boolean
h_div*	h is a divider (or a multiplier) in the weight smoothness regularisation	Boolean
optimiser*	Optimiser to use ("SGD" or "Adam")	String
momentum	SGD: Momentum value Adam: tuple containing β_1 and β_2	float (float, float)
crop_and_flip	Enable cropping and flipping of the training dataset	Boolean

Algorithm 1 Learning Process

```
1: procedure TrainNetwork(net, dataset_name, LearningParameters (abbreviated LP))
2:     $data \leftarrow load\_data(dataset\_name,\ LP.crop\_and\_flip)$
3:     $epochs \leftarrow (LP.training\_steps\_max * LP.batch\_size)/sizeOf(data)$
4:     $lr \leftarrow LP.lr$
5:     for all epochs do
6:         for all batches do
7:             $gradients \leftarrow 0$
8:             $output \leftarrow net(batch\,input)$
9:             $loss \leftarrow CrossEntropyLoss(output, batch\,target)$
10:            $loss \leftarrow loss + smoothing\,regularisation(net,\ LP.alpha,\ LP.b\_in\_reg,\ LP.h\_div)$
11:            $loss \rightarrow backward\,pass()$
12:            $optimiser(lr, LP.momentum, LP.wd) \rightarrow update(net)$
13:        end for
14:        if $current\,epoch$ is in $LP.lr\_decay\_at$ then
15:            $lr \leftarrow lr * LP.lr\_decay$
16:        end if
17:    end for
18: end procedure
```

3.2 Network Structure

The network structure is inspired by the structure proposed by Chang et al. [2]. The whole structure is presented in figure 3.1. First, the image received by the network goes through a convolutional layer in order to increase the number of channels. Then, if enabled, batch normalisation and then dropout are applied. Next, the data goes through M *units*. In each *unit*, a Hamiltonian Layer (described in the next section) is applied. Then the data dimension is reduced by pooling, and, finally, channels full of zeros are padded. Note that the last *unit* does not have a padding layer (we do not want to add inconsistent information before the classification). The output of the last unit then is reshaped to a vector. If enabled, the vector goes through another dropout layer. Finally, the data is classified by one (or two) fully connected (FC) layers.
As for the learning, the different options and parameters can be selected through a structure, *NetworkParameters*. The network will be directly created from it. The options are briefly presented in table 3.2. For the ones with an asterisk, more information is available in appendix A.2.2.

3.3 Hamiltonian Layer

As a reminder, a Hamiltonian layer starts each *unit* of the structure (green rectangles in figure 3.1). This layer consists of calling successively a certain number of Hamiltonian (each one taking as input the output of the previous one). Three different Hamiltonian are available: the Two-Layers (Chang et al. [2]), the J1 and the J2 (Galimberti et al. [3]). Their structure and their definition have been presented and discussed in section 2.2. All the information about this layer is defined in the *HamiltonianParameters* structure, which is presented in table 3.3. Again, parameters noted with an asterisk are further developed in appendix A.2.3.

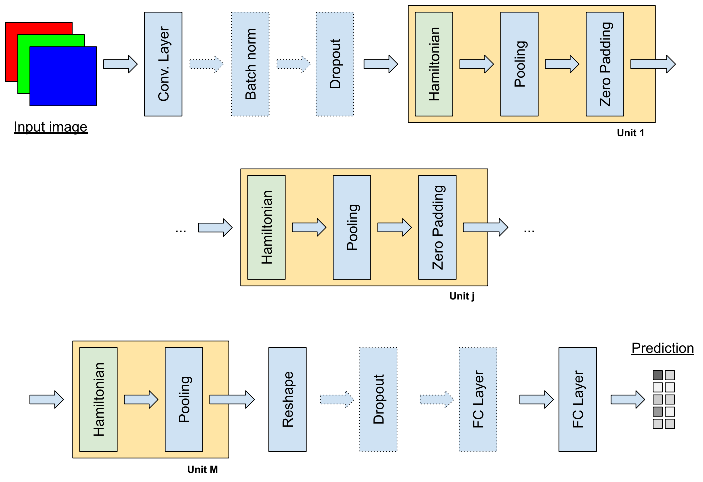


Figure 3.1: General network structure. Dashed layers are optional.

Table 3.2: Explanation of *NetworkParameters* parameters. Together, they define the structure of the network. Parameters noted with an asterisk are further explained in appendix A.2.2.

Parameters	Description	Data type
hamParams	Structure that defines the Hamiltonian Layer (see section 3.3)	*HamiltonianParameters*
nb_units	Number of units in the network	int
nb_features*	Desired number channels at the input of each *unit*	list
ks_conv	Kernel size of the convolutional layer	int
strd_conv	Stride of the convolutional layer	int
pd_conv	Zero-padding of the convolutional layer	int
pooling	Pooling method to use (Average ("Avg") or Maximum ("Max"))	str
ks_pool	Kernel size of the pooling layers	int
strd_pool	Stride of the pooling layers	int
init*	Initialisation method for the convolutional and FC layers	str
second_final_FC*	Additional FC layer. *None* to disable.	int
batchnorm_bool	Enable or not the batch normalisation layer	Boolean
both_border_pad*	"Sandwiches" the data with the new zeros channels if *True*	Boolean
dropout*	Dropout probabilities	list

Table 3.3: Explanation of *HamiltonianParameters* parameters. Together, they define the Hamiltonian to use in the network. Parameters noted with an asterisk are further explained in appendix A.2.3.

Parameters	Description	Data type
hamiltonianFunction	Hamiltonian to use ("TwoLayersHam","J1" or "J2")	str
n_blocks	Number of successive Hamiltonians	int
ks	Kernel size of the **K** convolutional operator	int
h	Discretisation timestep of the Hamiltonian	float
act	Activation function to use ("ReLU" or "Tanh")	str
init*	Initialisation method for **K** and **b**	str

4. Results

This chapter presents the results obtained using the framework introduced in the last chapter. First, the datasets used to evaluate it are presented. Then the performance achieved by the different H-DNN are shown. Finally, some other Image Classification Networks are described.
In this chapter, when not specified, accuracy refers to the testing accuracy on CIFAR-10. Most of the results have only been run once: this is justified by the very low standard deviation of the main results. All the computations have been done on the laboratory computer (briefly discussed in appendix A.3).

4.1 Datasets presentation

To evaluate the different networks, the CIFAR-10 dataset is used. This dataset contains 50'000 training images and 10'000 testing images. The images have a 32x32 resolution and are in colour. Each image belongs to one label over the ten existing ones (mainly animals or means of locomotion). Some examples are presented in figure 4.1.
Two others datasets are used to see if the networks can be extended to a different problem. The first one is CIFAR-100, which consists of the same images as CIFAR-10 but with one hundred classes (types of animals and means of locomotion are more detailed). The second one is STL-10. It contains 5'000 training images and 8'000 testing images. Images have a 96x96 resolution and are also in colour. There are, in total, ten classes of images. Examples from this dataset are presented in figure 4.2. The first supplementary dataset will test if the network can extend to datasets with more labels, and the second if the networks can learn with less but higher quality images.
The prepossessing steps for the three datasets are the same that have been proposed by Chang et al. [2]. For the two CIFAR datasets, first, 4 zeros are padded around the images. Then, random cropping that returns to the original image dimension is applied. Next, the images are randomly horizontally flipped. Finally, each channel is standardised by removing the mean and dividing by the standard deviation. The same operations are used for STL-10, but 12 zeros are padded instead of 4.

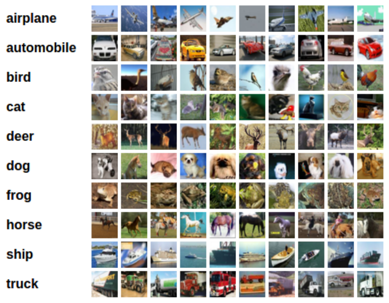


Figure 4.1: Example of CIFAR-10 images with their label. Image source: Pytorch

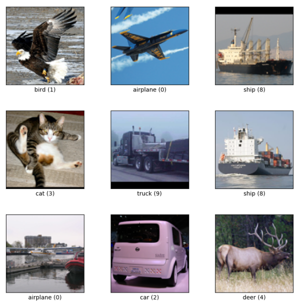


Figure 4.2: Example of STL-10 images with their label. Image source: TensorFlow

4.2 Chang et al. Two-Layers Hamiltonian

In first place, the network proposed b< Chang et al. [2] has been replicated. It has been discussed in section 2.3 and was represented in figure 2.1.
The network uses Two-Layers Hamiltonian blocks and is composed of 3 *units* with 6 Hamiltonian *blocks* each one. ReLU activation is used. The first unit receives 32 channels, the second 64 and the last one 112. The learning method lasts for 80'000 training steps and uses batches of 100 images. The learning rate starts at 0.1 and is divided by 10 at 80, 120 and 160 epochs. The smoothness regularisation is the one they presented in the paper (equation 2.17).
The structures used to define this network in the framework are presented in table 4.1. In this table, the parameters in italic had to be assumed due to a lack of information. Thus, they may not correspond to what Chang et al. [2] used. The assumptions made and their justifications are presented in Appendix A.4. Note that some of them are strong assumptions, which may result in performance difference (for example, the discretisation time (h) and the kernel size of $\mathbf{K}$ (ks)).
This network, that has **497'002 parameters**, achieves on **CIFAR-10** a testing accuracy of $\mathbf{85.52 \pm 0.23\%}$ and a training performance of $87.84 \pm 0.21\%$. This is not as good as what was claimed by Chang et al. [2]: 92.76%. The evolution of the accuracy and loss along epochs is presented in figure 4.3.
On **CIFAR-100** the network reaches $\mathbf{62.68 \pm 0.38\%}$ of testing accuracy and $71.19 \pm 0.27\%$ of training accuracy. Again, it smaller than what is claimed by Chang et al. [2]: 69.78%. The evolution of the accuracy and loss along epochs is presented in figure 4.12.
The network was also tested on another dataset, **STL-10**. Chang et al. [2] suggest to slightly modify their network for this new dataset: batch size is set to 128, the decay's weight to $5 * 10^{-4}$, the smoothness' weight to $3 * 10^{-4}$ and the maximum training steps to 20'000. To make it work h had to be set to 0.1 and the learning rate to 0.01. The decay epochs they gave led to poor results and thus [230, 350, 450] are used instead. All this results in a performance of **69.96%** on the testing dataset and of 80.15% on the training dataset. This performance is lower than the one they claimed: 85.5%.

Table 4.1: *Dataclasses* defining the network proposed by Chang et al. [2]. The parameters in italic where not clearly defined. The assumption made are explained and justified in Appendix A.4.

LearningParameters	
Parameters	Value
training_steps_max	80'000
batch_size	100
lr	0.1
lr_decay_at	[80,120,160]
lr_decay	0.1
wd	$2 * 10^{-4}$
alpha	$2 * 10^{-4}$
b_in_reg	False
h_div	True
optimiser	SGD
momentum	0.9
crop_and_flip	True

NetworkParameters	
Parameters	Value
hamParams	(right column)
nb_units	3
nb_features	[3, 32, 64, 112]
ks_conv	3
strd_conv	1
pd_conv	1
pooling	"Avg"
ks_pool	2
strd_pool	2
init	"Xavier"
second_final_FC	None
batchnorm_bool	False
both_border_pad	False
dropout	None

HamiltonianParameters	
Parameters	Value
hamiltonianFunctions	"TwoLayersHam"
n_blocks	6
ks	3
h	0.05
act	"ReLU"
init	"Xavier"

4.3 Enhanced Two-Layers Hamiltonian

4.3.1 Network specificationsn

In order to reach a comparable performance to the one claimed by Chang et al. [2], the previous network has been enhanced. For this aim, grid searches over the existing parameters have been done and additional layers have been implemented. This solution will be called "nominal" or "vanilla" Two-Layers Hamiltonian.
Table 4.2 sums up all the parameters of this nominal network. The parameters in bold indicate the differences with the original Chang et al. [2] network. In terms of learning, the last learning rate decay has been moved from 160 to 150 (160 being the last epoch, it is believed that for the decay to have a real impact, it should be used at least during 10 epochs). The weight smoothness regularisation has also been modified: b now contributes to it (as suggested by Haber and Ruthotto [1] and Galimberti et al. [3]) and the discretisation timestep h multiplies the smoothness loss (as suggested by Galimberti et al. [3]) instead of dividing it. New layers have been used in this network. Batch normalisation is applied just after the initial convolutional layer: a cleaner input to the H-DNN may give better results. Dropout is also added in two places to avoid overfitting: just after the first convolutional layer and just before the classification FC layer. Additionally, the "sandwich" padding method is used instead of appending to the end of the existing channels. Due to all these modifications, a new h value has been computed.
These enhancements result in higher performance for this network than for the previous one while having a comparable number of parameters: **497'066 parameters**. For **CIFAR-10** it achieves: **$91.58 \pm 0.14\%$** on the testing dataset and $98.91 \pm 0.10\%$ on the training dataset. The evolution of the performance and loss of this network is represented in figure 4.3.
To ensure that the nominal network could not be improved further, the network has also been trained with 100'000 training steps instead of 80'000. To use the additional epochs with full efficiency the learning rate decay epochs where adapted too ($lr\_decay\_at = [100, 150, 175]$). It results in 91.63% accuracy. So, more training steps do not substantially improve the performance, ensuring that this version of the network has reached a (local?) optimum.

Table 4.2: *Dataclasses* defining the nominal/vanilla Two-Layers Hamiltonian Network. The parameters in bold are the ones diverging from what was taken (or assumed) from Chang et al. [2].

LearningParameters	
Parameters	Value
training_steps_max	80'000
batch_size	100
lr	0.1
lr_decay_at	[80,120,150]
lr_decay	0.1
wd	$2 * 10^{-4}$
alpha	$2 * 10^{-4}$
b_in_reg	True
h_div	False
optimiser	SGD
momentum	0.9
crop_and_flip	True

NetworkParameters	
Parameters	Value
hamParams	(right column)
nb_units	3
nb_features	[3, 32, 64, 112]
ks_conv	3
strd_conv	1
pd_conv	1
pooling	"Avg"
ks_pool	2
strd_pool	2
init	"Xavier"
second_final_FC	None
batchnorm_bool	True
both_border_pad	True
dropout	[0.1,0.2]

HamiltonianParameters	
Parameters	Value
hamiltonianFunctions	"TwoLayersHam"
n_blocks	6
ks	3
h	0.2
act	"ReLU"
init	"Xavier"

4.3.2 Network analysis

The different hyperparameters of the network are analysed. All the following is based on the nominal Two-Layers Hamiltonian Network defined in table 4.2. Only the indicated parameters are modified from it. All the tests will be done on the CIFAR-10 dataset. As a reminder, **the nominal network achieved 91.58% of accuracy with 497'066 parameters**.
The hyperparameters are divided into groups depending on how many values have been tested for them. First, the "binary" are presented (the ones that have been either activated or not). Then the "few-cases" are studied (hyperparameters for which two or three values have been tested). Finally, the "grid-searched" hyperparameters analysis is shown.

4.3.3 Binary hyperparameters

- ***b_in_reg = False*** $\rightarrow$ **91.25%**: removing the smoothness regularisation of **b** penalises the network.
- ***pooling = "Max"*** $\rightarrow$ **87.68%**: using Maximum Pooling instead of Average Pooling reduces the capacity of the network. For this network to learn, the learning rate and the timestep have been changed: $lr = 0.05$ and $h = 0.05$.
- ***batchnorm_bool = False*** $\rightarrow$ **91.56%**: removing the batch normalisation layer does not harm the performance significantly. To ensure convergence, the learning rate used here is smaller ($lr = 0.05$).
- ***both_border_pad = False*** $\rightarrow$ **91.45%**: appending the channels full of zeros at the end of existing channels instead of around them slightly decreases the performance.
- ***crop_and_flip = False*** $\rightarrow$ **86.85%**: not cropping and flipping the images worsen the performance. More, the network reaches around 100% training accuracy around the 80^{th} epoch.
- ***init = "Normal"*** $\rightarrow$ **85.41%**: the normal initialisation is hard to make work. The default normal (mean=0, std=1) is unable to learn. Using another normal (mean=0, std=0.01) and by decreasing the learning rate, the discretisation time and changing the learning rate decay ($lr = 0.01$, $h = 0.05$ and $lr\_decay\_at = [120, 140, 150]$) the network was able to achieve the aforementioned performance. (Note that this initialisation was used for **K**, **b**, the convolutional and the FC layers.)
- ***init = "Ones"*** $\rightarrow$ **0.0%**: it was impossible to train the network with this initialisation. (Note that this initialisation was used for **K**, **b**, the convolutional and the FC layers.)
- ***act = "Tanh"*** $\rightarrow$ **81.24%**: using the Tanh activation function instead of ReLU inside of the Hamiltonian Layers shows weaker performance.
- ***second_final_FC = 100*** $\rightarrow$ **91.99%**: adding a second FC layer that has 100 units at the end of the network improves the performance. However, this improvement has a cost in terms of number of parameters: this network has 659'446 parameters.

4.3.4 Few cases tested hyperparameters

Kernel size (ks) of the convolutional operator:

1. $ks = 1$: 76.29% with 73'130 parameters;
2. $ks = 3$: 91.58% with 497'066 parameters;
3. $ks = 5$: 91.28% with 1'344'938 parameters (learning rate reduced to 0.05).

A kernel size bigger than 1 gains a lot in performance. But, increasing it even more shows no improvement.
Number of units (*nb_units*):

1. *nb_units* $= 2$: 90.64% with 497'738 parameters (number of channels: $3- > 56- > 112$) (learning rate is set to 0.05 and h to 0.1);
2. *nb_units* $= 3$: 91.58% with 497'066 parameters (number of channels: $3- > 32- > 64- > 112$);
3. *nb_units* $= 4$: 92.01% with 783'458 parameters (number of channels: $3- > 36- > 70- > 112$).

The deepness of the network improves the performance.
Note that for each case, the last unit receives 112 channels (thus, this can not interfere with the hyperparameter that is being studied). The number of parameters is the same for 2 or 3 units because the number of parameters in the classifier is bigger in the 2 units case (the images have only be pooled one time).

Number of Hamiltonian Blocks (*n_blocks*) in the *unit*:

1. *n_blocks* $= 6$: 91.58% with 497'066 parameters;
2. *n_blocks* $= 18$: 92.11% with 1'453'418 parameters (h reduced to 0.1).

Again, the increase of deepness increases the performance.
Dropout (*dropout*):

1. *dropout* $= [0, 0]$: 91.31% on testing and 99.31% on training;
2. *dropout* $= [0.1, 0.2]$: 91.58% on testing and 98.91% on training;
3. *dropout* $= [0.2, 0.5]$: 91.63% on testing and 98.29% on training.

No dropout results in higher training performance but lower testing performance. Some dropout increases the testing performance while diminishing the training performance. But increasing even more the dropout does not increase more the testing performance.
As a reminder, the first value indicates the dropout after the initial convolutional layer and the second the dropout after the last *unit* (before classification).

4.3.5 Grid searched hyperparameters

The discretisation time, h, is a crucial parameter to analyse via grid search. The grid search has been done in both the case where h divided the smoothing regularisation term and the case where it multiplies it (see appendix A.2.1 for more details). The results are shown in figure 4.4.
The plot shows that there seems to be an optimum around 0.1 for both methods. Multiplying by h gives better performance for any h. With too high values the network does not learn and with too low ones the networks performs more poorly.

The regularisation terms that multiply the weight decay regularisation (*wd*) and the weight smoothing regularisation (*alpha*), are studied through a grid search where they share the same value. The result is presented in figure 4.5.
The plot shows that with too small values, the network cannot learn and that too high values lead to poor performance. It looks like there is an optimum at 10^{-4}.

The learning rate is another important hyperparameter. Its influence is shown in figure 4.6. Too high learning rate prevents the network from learning, whereas too small one lead to slower and poorer performance. 10^{-1} is an optimum.

The momentum is the hyperparameter of the stochastic gradient descent (SGD). Its grid search is plotted in figure 4.7. From the plot, it seems that it would have been optimal to use 0.7 instead of 0.9.

The Adam optimiser has also been studied (it is described in appendix A.2.1.3). A new learning rate has to be researched for this new optimiser and its hyperparameters β_1 and β_2 had to be studied.
The optimal learning rate is 0.001 based on plot 4.8. Figure 4.9 shows that the optimal β_1 is 0.7 and from figure 4.10 β_2 should be 0.7 as well.
The fusion of these three value results in results in 88.86% accuracy when not using learning rate decay. When also using the decays, it achieves 91.14%. The evolution in accuracy along epochs of both networks is plotted in figure 4.11 along with the nominal network.

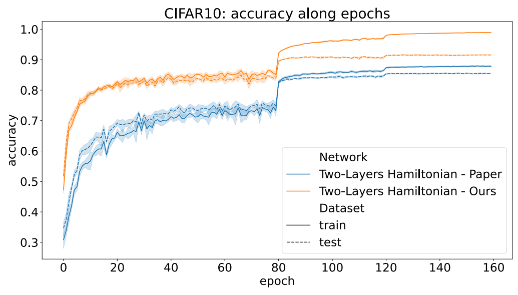
4.3.6 Other datasets

The performance of the vanilla network when facing a subset of the CIFAR-10 dataset has been evaluated. It was done with 20%, 15%, 10%, 5% and 1% of the data. The discretisation time had to be lowered for the two last values to ensure learning ($h = 0.1$). Note that because the dataset is smaller, the number of epochs is higher. Consequently, the epochs at which the learning rate is decayed have been adapted (while keeping the same proportion).
Figure 4.13 shows the performance obtained. With only 10% the network still performs well (more than 80%). Even with 1% of the data, the network still does significantly better than a random classifier. The results of this network facing reduced data is comparable to what Chang et al. [2] presented and even seems to show better performance for very few data (see fig. 2.3).

The influence of a higher number of labels has also been inspected by testing the nominal network on **CIFAR-100**. The obtained accuracy is $\mathbf{70.23 \pm 0.19\%}$ on the testing dataset and $90.14 \pm 0.19\%$ on the training dataset. The evolution of the performance is represented in figure 4.12.

The network has also been evaluated on a completely different dataset: **STL-10**. On the network side, only the time discretisation has been decreased ($h = 0.01$). On the learning side, in the same spirit as what Chang et al. [2] did, the batch size has been set to 128, the number of training iterations to 20'000, the learning rate is set to 0.05 and the learning rate decays happen at 300, 400 and 450 epochs. The performance obtained is **80.33%** on testing and 99.6% on the training dataset.

Hamiltonian Network - Two Layers



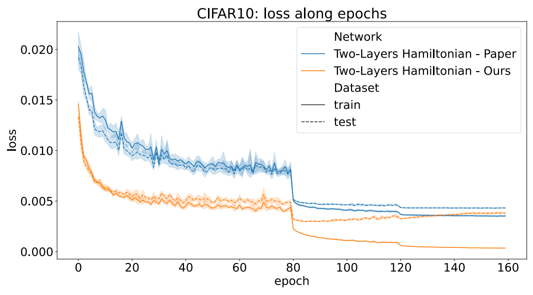


Figure 4.3: Evolution of the loss and accuracy along epochs of Two-Layer Hamiltonian based networks on CIFAR-10. In blue, our implementation of the network from Chang et al. [2] and in orange our nominal network. Note that the training has been repeated 5 times: the line indicates the mean performance and the gray area the standard deviation.

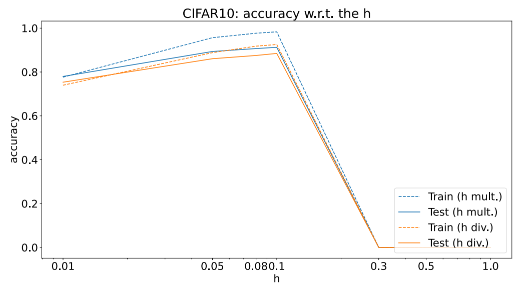


Figure 4.4: Influence of h on the performance of the nominal network on the CIFAR-10 dataset. In the orange case, h divides the smoothing regularisation term and in the blue case it multiplies it.

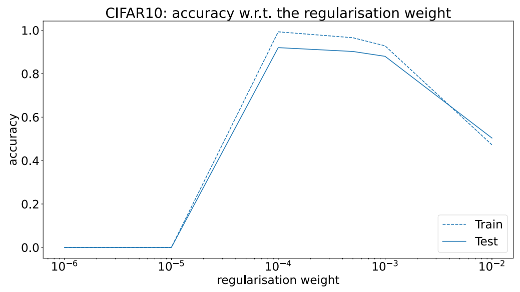


Figure 4.5: Influence of the weight of the regularisation terms (smoothing $(alpha)$ and weight decay (wd)) on the performance of the nominal network on the CIFAR-10 dataset.

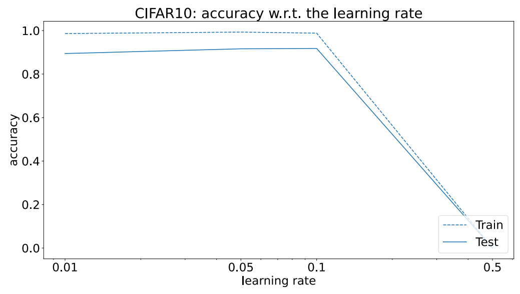


Figure 4.6: Influence of the learning rate on the performance of the nominal network on the CIFAR-10 dataset.

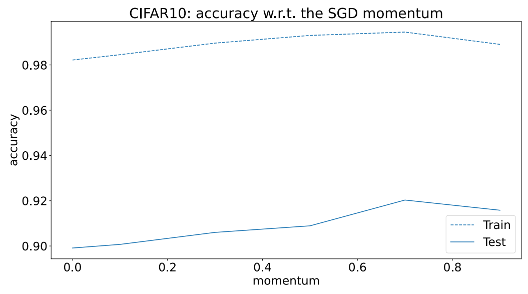


Figure 4.7: Influence of the momentum of the Stochastic Gradient Descent on the performance of the nominal network on CIFAR-10 dataset.

Hamiltonian Network - Two Layers - Adam Optimiser

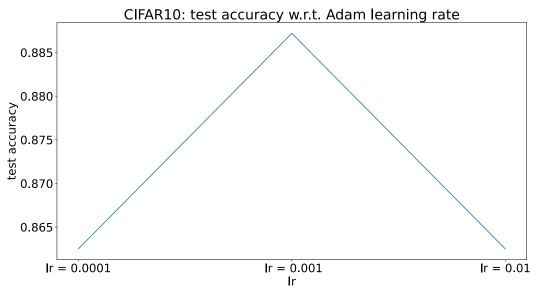


Figure 4.8: Influence of learning reate on the performance of the Adam optimised nominal network on the CIFAR-10 dataset.

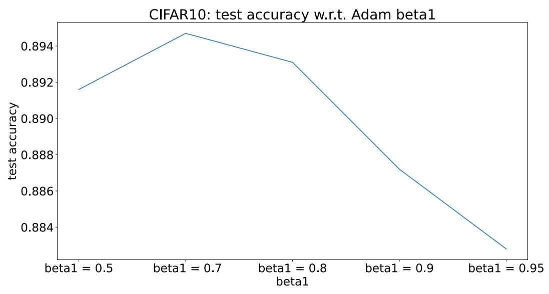


Figure 4.9: Influence of β_1 on the performance of the Adam optimised nominal network on the CIFAR-10 dataset.

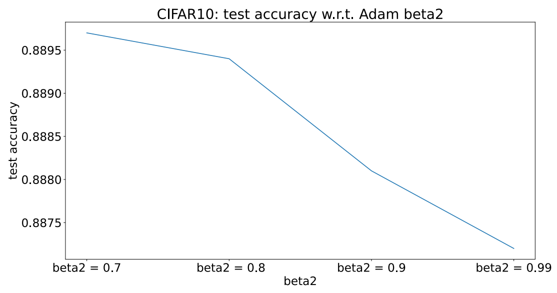


Figure 4.10: Influence of β_2 on the performance of the Adam optimised nominal network on the CIFAR-10 dataset.

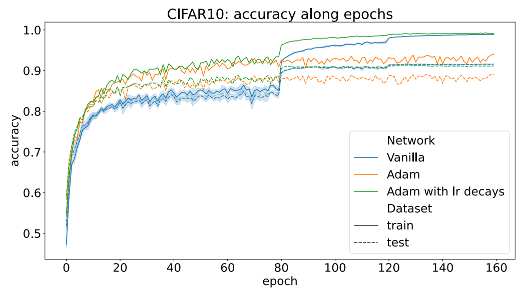


Figure 4.11: Evolution of the training and testing accuracy on CIFAR-10 of the nominal network (blue), the Adam optimised nominal network (orange) and the Adam optimised nominal network with learning rate decays (green).

Hamiltonian Network - Two Layers - Modified/Other dataset

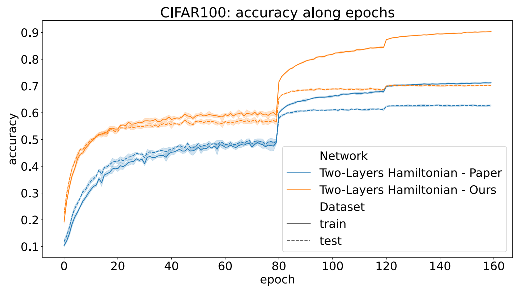


Figure 4.12: Accuracy along epochs of Two-Layer Hamiltonian networks on CIFAR-100. In blue our implementation of the Chang et al. [2] network and in orange our nominal network. Training has been repeated 5 times: the line indicates the mean performance and the gray area the standard deviation.

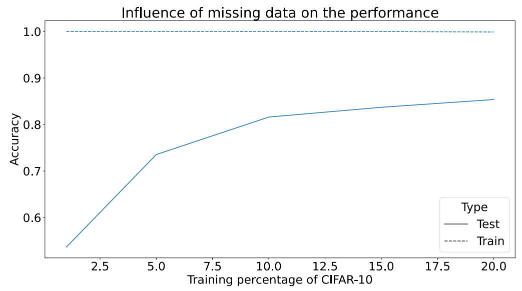


Figure 4.13: Performance obtained by the nominal network when dealing with a reduced CIFAR-10 dataset.

4.4 J1 Hamiltonian

The framework has also been tested using the J1 architecture. It has been presented in table 2.1 using the unified framework notation [3] (equ. 2.20 & 2.21). It is used both with ReLU and Tanh activations.
Note that the plots for this network are in the appendix (section A.5) because their global shape is qualitatively (but not quantitatively) similar to the ones of the nominal network that were already shown.

4.4.1 ReLU

First, the J1 architecture has been tested with the ReLU activation function. All the parameters describing this network are presented in 4.3. The differences with the nominal network are printed in bold. Only the Hamiltonian function, the discretisation time (h), the learning and its decays had to be modified. This network has **973'944 parameters**
The final performance obtained by this network on CIFAR-10 is **92.27%** on the testing dataset and 99.15% on the training one. The evolution of its performance and loss is presented in the appendix in figure A.3.
A smaller version of it has been implemented. Instead of 3 *units* the network has 2 *units*. The features table ($nb\_features$) is now [3, 40, 80]. For this smaller version, the discretisation time (h) is reduced to 0.05. This smaller version has **485'130 parameters**. It achieves **90.4%** on the testing dataset and 94.69% on the training one despite having half of the parameters. Its accuracy evolution along epochs is plotted on the same figure.

The grid searches along the discretisation time (h), the regularisation weights (same value for weight decay (wd) and weight smoothing ($alpha$)), the learning rate (lr) and the dropout intensity ($dropout$) are presented in appendix in figures, respectively: A.4, A.5, A.6 and A.7.

Table 4.3: *Dataclasses* defining the J1-ReLU Hamiltonian Network. The parameters in bold are the ones diverging from the nominal network.

LearningParameters	
Parameters	Value
training_steps_max	80'000
batch_size	100
lr	0.05
lr_decay_at	[120, 140, 150]
lr_decay	0.1
wd	$2 * 10^{-4}$
alpha	$2 * 10^{-4}$
b_in_reg	True
h_div	False
optimiser	SGD
momentum	0.9
crop_and_flip	True

NetworkParameters	
Parameters	Value
hamParams	(right column)
nb_units	3
nb_features	[3, 32, 64, 112]
ks_conv	3
strd_conv	1
pd_conv	1
pooling	"Avg"
ks_pool	2
strd_pool	2
init	"Xavier"
second_final_FC	None
batchnorm_bool	True
both_border_pad	True
dropout	[0.1,0.2]

HamiltonianParameters	
Parameters	Value
hamiltonianFunctions	"J1"
n_blocks	6
ks	3
h	0.1
act	"ReLU"
init	"Xavier"

4.4.2 Tanh

The J1-Tanh network architecture is specified by the three structures presented in 4.4. The structure from the nominal network is mainly kept. The differences with it are printed in bold. The network parameters are the same. In the Hamiltonian parameters, only the Hamiltonian and the activation functions are changed. The learning parameters differ in terms of total number of training step, learning rate, learning decay epochs and regularisation weights. This network has **973'944 parameters**

The final performance obtained by this network on CIFAR-10 is **89.07%** on the testing dataset and 97.89% on the training one. The evolution of its performance and loss is presented in the appendix in figure A.8.
A reduced version of this network has been implemented. Instead of 3 *units* the network has 2 *units*. The features table ($nb\_features$) is now [3, 40, 80]. Nothing other has been modified. This smaller version has **485'130 parameters**. It achieves **87.03%** on the testing dataset and 92.69% on the training dataset. Its evolution is plotted on the same figure.

The grid searches along the discretisation time (h), the regularisation weights (same value for weight decay (wd) and weight smoothing ($alpha$)), the learning rate (lr) and the dropout intensity ($dropout$) are presented in appendix in figures, respectively: A.9, A.10, A.11 and A.12.

Table 4.4: *Dataclasses* defining the J1-Tanh Hamiltonian Network. The parameters in bold are the ones diverging from the nominal network.

LearningParameters	
Parameters	Value
training_steps_max	100'000
batch_size	100
lr	0.01
lr_decay_at	[120,160,190]
lr_decay	0.1
wd	$2 * 10^{-6}$
alpha	$2 * 10^{-6}$
b_in_reg	True
h_div	False
optimiser	SGD
momentum	0.9
crop_and_flip	True

NetworkParameters	
Parameters	Value
hamParams	(right column)
nb_units	3
nb_features	[3, 32, 64, 112]
ks_conv	3
strd_conv	1
pd_conv	1
pooling	"Avg"
ks_pool	2
strd_pool	2
init	"Xavier"
second_final_FC	None
batchnorm_bool	True
both_border_pad	True
dropout	[0.1,0.2]

HamiltonianParameters	
Parameters	Value
hamiltonianFunctions	"J1"
n_blocks	6
ks	3
h	0.2
act	"Tanh"
init	"Xavier"

4.5 J2 Hamiltonian

Another Hamiltonian architecture is tested, the J2 Hamiltonian. It has been presented in table 2.1 using the unified framework notation [3] (equ. 2.20 & 2.21). J2 will mainly be studied using ReLU activation. Again, the plots of this network can be found in the Appendix (section A.5).
The parameters defining it are shown in table 4.5. Parameters in bold are the ones that have been modified from the nominal ones. Only the Hamiltonian function and its discretisation time (h) have been modified regarding the network structure. Regarding the learning problem, the learning rate is lower, there are more training steps and the learning rate decay occur at different epochs. This network has **973'944 parameters**
The final performance obtained by this network on CIFAR-10 is **87.64%** on the testing dataset and 95.62% on the training one. The evolution of its performance and loss is presented in the appendix in figure A.13.
A reduced version of this network has been implemented. Instead of 3 *units* the network has 2 *units*. The features table ($nb\_features$) is now [3, 40, 80]. The learning rate had to be lowered to 0.005, the number of iteration augmented to 150'000 and the learning decays now happen at [200,250,275]. This smaller version has **485'130 parameters** and achieves **85.63%** on the testing dataset and 90.5% on the training one. Its evolution is plotted on the same figure.

The network has been evaluated with a **Tanh** without any other modification. On CIFAR-10, it achieved **84.59%** of accuracy on the testing dataset and 88.40% on the training one. The evolution of the accuracy along epochs is available in the appendix in figure A.17.

The grid searches along the discretisation time (h), the regularisation weights (same value for weight decay (wd)) and weight smoothing ($alpha$) and the learning rate (lr) are presented in appendix in figures, respectively: A.14, A.15 and A.16.

Table 4.5: *Dataclasses* defining the J2 Hamiltonian Network. The parameters in bold are the ones diverging from the nominal network.

LearningParameters	
Parameters	Value
training_steps_max	100'000
batch_size	100
lr	0.01
lr_decay_at	[150, 180, 190]
lr_decay	0.1
wd	$2 * 10^{-4}$
alpha	$2 * 10^{-4}$
b_in_reg	True
h_div	False
optimiser	SGD
momentum	0.9
crop_and_flip	True

NetworkParameters	
Parameters	Value
hamParams	(right column)
nb_units	3
nb_features	[3, 32, 64, 112]
ks_conv	3
strd_conv	1
pd_conv	1
pooling	"Avg"
ks_pool	2
strd_pool	2
init	"Xavier"
second_final_FC	None
batchnorm_bool	True
both_border_pad	True
dropout	[0.1,0.2]

HamiltonianParameters	
Parameters	Value
hamiltonianFunctions	"J2"
n_blocks	6
ks	3
h	0.01
act	"ReLU"
init	"Xavier"

4.6 Alternative Networks

4.6.1 AlexNet

AlexNet has been introduced in 2012 by Krizhevsky et al. [4] . Its architecture is quite simple: five convolutional layers followed by three FC layers. It uses maximum pooling and ReLU activation. Additionally, dropout, weight decay, data augmentation and "response-normalisation" were used. A representation is given in figure 4.14. Please refer to the paper for more details.
Krizhevsky et al. [4] used a modified version of their network for the CIFAR-10 dataset classification, but I was not able to find the details about it. As a workaround, *PyTorch* provides a pre-built and pre-trained *AlexNet* for the other dataset. It has been assumed that the feature of these images may share a lot of common with the ones of CIFAR. Thus, the pre-trained network has been downloaded, and the last classification layer (an FC layer) has been modified to have the same number of outputs as the number of CIFAR labels. The network has be retrained during some epochs on CIFAR. The CIFAR images have been resized (by interpolation) to match the network's input. The other pre-processing steps used are centre cropping and normalisation.
The network has **57'044'810 parameters**. On CIFAR-10, it results in a testing accuracy of **91.55%** and a 100% training accuracy. Despite not being able to know exactly how they implemented *AlexNet* for CIFAR, this result makes sense because, in the paper, they claim a 89% of accuracy on CIFAR-10. Their performance may be smaller because their network for CIFAR-10 may also be smaller than this one.

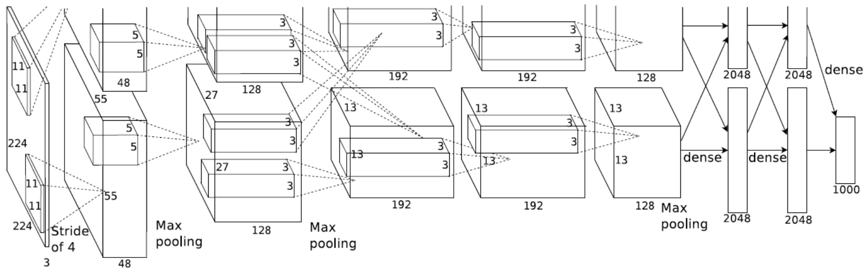


Figure 4.14: Illustrative representation of the *AlexNet* [4] (image from their paper). This image does not correspond to the network used for CIFAR-10, but the concept is similar.

4.6.2 ResNet

Another network has been implemented to deal with CIFAR-10 images: *ResNet*. It is much closer to H-DNN than *AlexNet* because H-DNN basic architecture is inspired by *ResNet* layers. This network was introduced in 2016 by He et al. [5]. The network starts with a convolutional layer. It is followed by pairs of residual convolutional layer (i.e. $y = x + F_1(F_2(x))$). Finally, the classification is done with a fully connected layer that uses Softmax activation. The activation function for the others layers is ReLU. Figure 4.15 shows the structure they used for another dataset (the idea is similar for CIFAR-10). Please refer to the paper for more details.
PyTorch provides a pre-built and pre-trained *ResNet*, but again it is the one used for another dataset. However, Idelbayev proposes on his *GitHub* [6] different pre-built and pre-trained *ResNet* implemented like proposed in the paper for CIFAR. His code has been verified and the result he claims correspond to the paper's result. The pre-processing steps are 4-zeros padding, random cropping, random flipping and normalisation.
The performance and the number of parameters depend on the number of layers of the network. Table 4.6 sums up this information. These results are close to what the paper claims [5].

Table 4.6: ResNet performance on CIFAR-10 and number of parameters for different total number of layers.

# of layers	20	32	44	56	110	1202
# of parameters	269'722	464'154	658'586	853'018	1'727'962	19'421'274
Train accuracy (%)	99.71	99.94	99.98	99.97	>99.99	>99.99
Test accuracy (%)	**91.73**	**92.63**	**93.1**	**93.39**	**93.68**	**93.82**

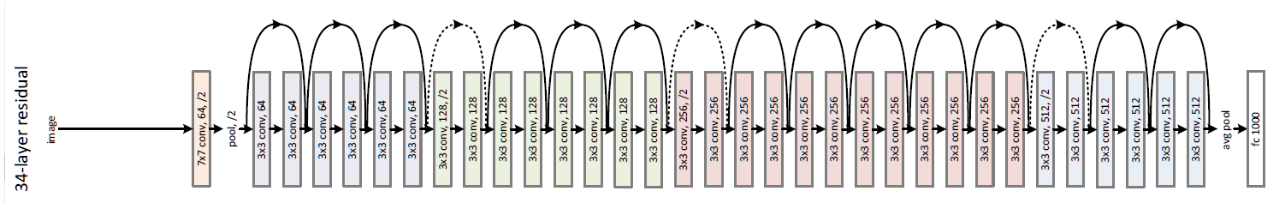


Figure 4.15: Illustrative representation of *ResNet* [5] (image from their paper). This image does not correspond to the network used for CIFAR-10, but the concept is similar.

4.6.3 Others

The current state of the art achieves high accuracy. Most of them use many parameters and are complex. Thus, these networks were not implemented and are just briefly presented along with their performance.

Wikipedia on its CIFAR-10 page refers to **GPipe** as the highest test accuracy state of the art network. This network was introduced by Huand et al. [7] in 2019. Interestingly, the paper does not provide a new network but rather introduce a novel way to train giant networks. Using a novel batch-splitting pipe-lining they have been able to train a **557 million parameters** *AmoebaNet* [8] and obtained **99.0% accuracy on CIFAR-10** and **91.3% accuracy on CIFAR-100**. *AmoebaNet* is a network which automatically evolves.

The higher CIFAR-10 accuracy paper when referring to *benchmarks.ai* is **Big Transfer** (BiT-L). It was introduced by Kolesnikov et al. [9] in 2019. In the paper, they revisit the transfer learning by giving some rules about how to train the *upstream* network and how transfer its knowledge to the *downstream* network. Using *Big Transfer* they reached **99.37% of accuracy on CIFAR-10** and **93.51% on CIFAR-100**. The upstream network has **928 million parameters**.

paperswithcode.com higher accuracy paper on CIFAR-10 is **EffNet-L2 + SAM**, presented in 2020 by Foret et al. [10]. The proposed novelty is to minimise the loss value as well as the loss sharpness. A low loss sharpness means that parameters in the neighbourhood also have a low loss. Applying this new loss on *EfficientNet-L2* increases its performance up to **99.7% of accuracy on CIFAR-10** and **96.08% on CIFAR-100**. EfficientNet (EffNet) is a semi-supervised learning network [11]: the network is given labelled image as well as pseudo labelled images (labels assessed by a "teacher"). The final network has **480 million parameters**.

4.7 Summary

Table 4.7: Summary of the performance and shapes of different networks that have been presented. Networks are grouped by number of parameters. Best accuracy of each group is printed in bold.

Model	# of layers	# of parameters (M)	Test accuracy (%)	Train accuracy (%)
Two Layers - Paper (theirs)	74	0.43	**92.76**	-
Two Layers - Paper (by us)	74	0.50	85.52	87.48
Two Layers - Ours	74	0.50	91.58	98.91
Reduced J1 - Tanh	38	0.49	87.03	92.69
Reduced J1 - ReLU	38	0.49	90.4	94.69
Reduced J2	38	0.49	85.63	90.5
ResNet-32	32	0.46	92.63	99.94
J1 - Tanh	56	0.97	89.07	97.89
J1 - ReLU	56	0.97	92.27	97.15
J2	56	0.97	87.64	95.62
ResNet-56	56	0.85	**93.68**	>99.99
AlexNet	8	57	91.55	100
ResNet-1202	1202	19.4	**93.82**	>99.99
GPipe	-	557	99.0	-
BiT-L	-	928	99.37	-
EffNet-L2 + SAM	-	480	**99.7**	-

5. Discussion

5.1 Hamiltonians Neural Networks

5.1.1 Performance

All the H-DNN achieved much higher performance than what a random classifier would achieve. Moreover, this performance can be quite easily transferred to other problems. The nominal Two-Layers Hamiltonian performed efficiently when dealing with more labels (CIFAR-100 instead of CIFAR-10 (fig. 4.12)). Its resistance to missing data is also quite impressive (fig. 4.13). Moreover, it worked with only a few modifications on another dataset, STL-10 (section 4.3.6). Additionally, the standard deviation of the performance is low (fig. 4.3 and 4.12). All this indicates that H-DNN can be trained efficiently using the implemented framework.

All the results were discussed in terms of global accuracy. Figure 5.1 presents the confusion matrix of the nominal Two-Layers Hamiltonian Network. Thanks to this matrix, we can see which labels are confused and if the network cannot learn a specific class. Most of the classes seem to have around the same accuracy except for the "cat" class, which has a lower one. The "cat" class is often confused with the "dog" class (it is the most common confusion). This confusion makes sense as the two animals share a common shape and mainly appear in the human environment (city, homes, ...). The confusion between "deer" and "horse" is also frequent and may be mainly due to the similarities between them, whereas the confusion between "bird" and "plane" may result from the common background that they often share (sky). The confusions between really different labels are rare: for example, "dog" and "truck" are only mistaken four times in total.

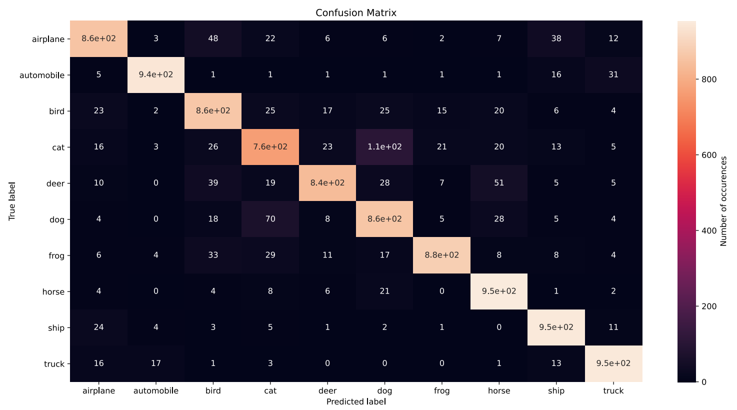


Figure 5.1: Confusion matrix of the nominal Two-Layers Hamiltonian Network.

A qualitative analysis of the performance can be done by looking at how an image evolves through the network. For this experiment, the nominal Two-Layers Hamiltonian Network will be used again. Only one interesting channel at each critical point will be represented. Note that most of what will follow are assumptions about what the network does: it is impossible to know which feature the network "wants" to extract and how it will benefit from it. More, some filters may have no sense for some images.
The evolution of the image of a horse and of a ship is studied in figures 5.2 and 5.3, respectively. For both, after the first convolutional layer, the image is still recognisable in some channels. In both cases, the selected channel seems to contain a representation of the edges of the image. After the first Hamiltonian unit, the horse is harder to recognise. It looks like the network did some blurring while still keeping some information about the edges (maybe to remove noisy information). After the first Hamiltonian Network, the ship is still well recognisable and even more complete than before. It can be supposed that the network used different edge detection method and combined them in this image to create a more reliable representation of the global shape. After the second unit, the selected channel for both the horse and the ship can be interpreted as a rough estimation of their shape (for the horse, it is possible to distinguish the ear, and for the ship, it is possible to see the masts). However, after the third Hamiltonian Unit, the data map makes no longer sense on any channels. The network may here encode high-level features used by the FC layer to predict the image's label. Both images are correctly classified. The second most possible class for the horse is "dog" and the least possible is "ship". For the ship, the second most probable class is "airplane" and the most unlikely is "deer".
The different channels that have been presented show that the network extracted features that a human can interpret and explain. However, most of the channels made little or no sense. Nevertheless, this does not mean that the network is misbehaving. To investigate further, the analysis of the different kernels could have been made.

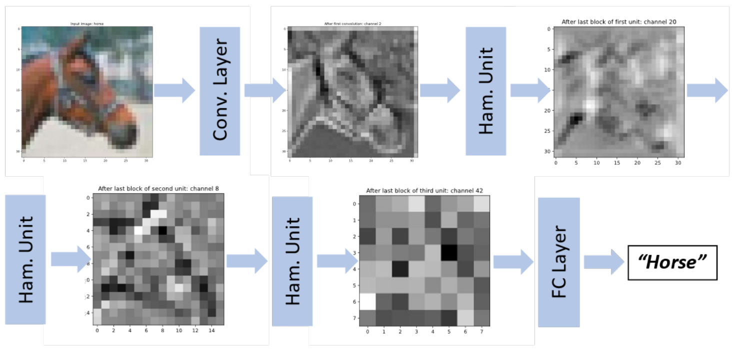

Input image: horse
Conv. Layer
After first convolution: channel 2
Ham. Unit
After last block of first unit: channel 20
Ham. Unit
After last block of second unit: channel 8
Ham. Unit
After last block of third unit: channel 42
FC Layer
"Horse"

Figure 5.2: Image transformation of a horse image by the nominal network.

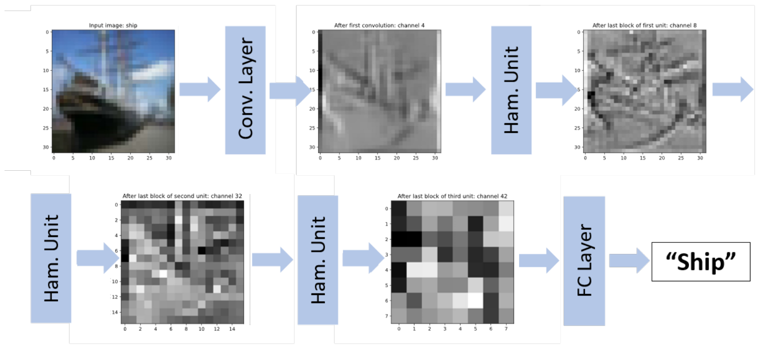

Input image: ship
Conv. Layer
After first convolution: channel 4
Ham. Unit
After last block of first unit: channel 8
Ham. Unit
After last block of second unit: channel 32
Ham. Unit
After last block of third unit: channel 42
FC Layer
"Ship"

Figure 5.3: Image transformation of a ship image by the nominal network.

5.1.2 Main hyperparameters

The H-DNN ensure that no gradient vanish or explosion occurs as long as their underlying assumptions are fulfilled. Thus, to ensure correct learning, three main parameters had to be correctly tuned: the discretisation time (h), the smoothing decay ($alpha$) and the learning rate (lr) .
First, the discretisation timestep h seems optimal when it is the biggest possible while respecting the assumption not to have a too big h. This can be seen on the different grid searches around this parameter (figs: 4.4, A.4, A.9 & A.14): too high value of h lead to the network to collapse and smaller values lead to weaker accuracy. Having a discretisation timestep too close to the instability leads to weakness when modifying another hyperparameter or facing a new dataset. So, to have a robust and efficient network, the ideal may be to take h slightly smaller than the most extreme value that it can be. Haber and Ruthotto [1] explained why the network collapses with high h. On the opposite, the network may have lower performance with smaller h because, for a fixed number of *blocks*, it results in less time to modify the data ($T = h * n_{blocks}$). Less modification means less feature extraction and thus less performance.
Second, the smoothing regularisation (scaled by $alpha$) is also a necessary condition of H-DNN. The grid searches over this parameter (figs: 4.5, A.5, A.10 & A.15) are not identical. However, the tendency indicates that for too small values the network is not able to learn (since the condition is not respected) and that with too high values the performance worsens (supposedly, the regularisation loss becomes too important with respect to the cross-entropy loss and thus the network is not able to learn the latter). Most of the grid-searches have a plateau in between the two extremes where the performance is quite stable.
Third, the learning rate lr has the classic influence that it has on neural networks: too low one leads to slower learning and too high leads to inability to learn and even to network collapse (figs: 4.6, A.6, A.11 & A.16). The network collapse may be explained by the smoothing condition not being respected: the network may change the weights of $\boldsymbol{K}$ too rapidly for the smoothing to prevent it.
All these three parameters are closely related: h and $alpha$ appear in the same equation for the smoothness regularisation and lr will influence how the $\boldsymbol{K}$ changes and so if the smoothness is fulfilled. Thus, they should be carefully investigated in a common grid-search and not separated like done here.

5.1.3 Hamiltonian Functions

Three different Hamiltonian Functions have been evaluated in this project: Two-Layers, J1 and J2. Two-Layers and J1 are more similar than J2. Their architecture is close (see table 2.1). The differences are the discretisation method and that $\boldsymbol{K}$ is free in J1. Thus, only a few parameters had to be changed between the nominal Two-Layers network and the J1-ReLU network. The additional freedom on $\boldsymbol{K}$ led J1-ReLU to perform better than the Two-Layers but uses more parameters (see sum-up table 4.7). It also does better than J2.
J2 being different, other hyperparameters have been selected. Notably, h has to be smaller, which is due to the completely different shape of $\boldsymbol{J}$. This other Hamiltonian performs poorly than the two others (see table 4.7). This is surprising because J2 performed better on MNIST in the Galimberti et al. paper [3]. Nevertheless, one has to remember that the whole framework architecture has been inspired by Chang et al. [2] for Two-Layers Hamiltonians. So, maybe a different structure has to be considered to benefit fully from the J2 architecture.

5.1.4 Activation Functions

Tanh and ReLU have been studied in the J1 Hamiltonian architecture. ReLU presented the best performance. This is also the case in the less in-depth evaluation of Tanh in the Two-Layers. Thus, ReLU seems to be a better option. Krizhevsky et al. [4] (*AlexNet* authors) say that ReLU layers are able to learn faster than Tanh. This is also the case with H-DNN, where a lower learning rate had to be used for Tanh (fig. A.3 and fig. A.8). They claim that this difference is because ReLU has non-saturating non-linearity compared to the saturating non-linearity of Tanh and Sigmoid. But, the difference between J2 using Tanh or ReLU is small.

5.1.5 Add-ons

The framework proposes some add-ons that were not included in the Chang et al. [2] structure. Most of them have a good influence.

- Adding ***b* in the smoothness regularisation** term, as suggested by Haber and Ruthotto [1] and Galimberti et al. [3] has a good influence (section 4.3.3). It makes sense as $\boldsymbol{b}$ has to change slowly for the H-DNN conditions to hold.
- Changing the pooling to **maximum pooling** reduces the performance (section 4.3.3). It may be due to the loss in information compared to average pooling since only one of the four pixel is considered.
- The **batch normalisation** which has been added just before the first Hamiltonian Layer does not increase significantly the performance (section 4.3.3) but, from experience, it allows more flexibility on the hyper-parameter selection: a standardised input is always easier to deal with.
- Implementing the **zero channels padding around the channels** instead of just appending them at the end improves slightly the performance (section 4.3.3). Otherwise, one of the "branch" (see figure 2.2) would receive only zeros.
- The **Xavier** and the **Xavier-like initialisation** for $\boldsymbol{K}$, $\boldsymbol{b}$, the convolutional layer and the FC layer really outperforms the other initialisation (Normal or Ones) (section 4.3.3). The efficiency of this initialisation is shown by Glorot et Bengio in their paper [12].
- The **additional fully connected layer** increases the performance (section 4.3.3) because the network gets better at the classification step. However, the cost in terms of parameters is too high to make it a competitive solution.
- The two **dropout** layers (one before the first Hamiltonian Layer and one before the classification) improve the performance by reducing the training overfit (section 3.2, fig. A.7 and fig. A.12). However, it adds two parameters that must be correctly selected to avoid accuracy decay due to information removal.
- The main hope with the **Adam optimiser** (described in appendix A.2.1.3) was that it would allow the user not to define the learning rate decays (Adam can automatically deal with more complex objective functions). Adam shows better performance than SGD without learning rate decays. However, it is not able to reach as good performance as SGD with the decays (fig. 4.11). Nevertheless, using the learning rate decays and ADAM can reach a comparable performance to SGD (same figure). Thus, Adam is still attractive because it will, in the first place, give better results than SGD and will still be able to reach the performance of SGD.

5.1.6 Network enlargement

There are different ways to increase the number of parameters of the network and increase its performance. Three different methods have been tested and evaluated in section 3.2.
First, the kernel size (ks) of $\boldsymbol{K}$ is set to 1, 3 and then 5. ks of 3 increases the performance of 15.29% for around 7 times more parameters. This significant improvement makes sense as the 3x3 kernel can benefit from the neighbourhood, whereas the 1x1 could not. However, increasing even more the kernel size (to 5) reduces the performance by 0.3% with respect to the solution that uses 3 as kernel size. That may be due to overfitting. But the most plausible cause is that this kernel size is too big for the final image size. Remember that, after each unit, a pooling that divides by two the data is applied. In the last unit, the images are 8x8 and so using a 5x5 kernel seems sub-optimal. It could be faced by resizing the images or applying less often the pooling.
Second, the number of *units* has been increased (while keeping the number of channels at the end of the

network identical). 3 *units* instead of 2 improved the performance by 0.94% for no increase in the number of parameters (more *units* results in smaller images due to pooling and thus to smaller classification FC layer). Going even further to 4 *units* increases again of 0.43% the performance for only half more parameters. This again highlights the importance of deepness in DL. Note that to add even more *units* one should face the pooling issue (images will be too small at some point).
Third, the number of *blocks* in each Hamiltonian units has been modified. From 6 to 18 *blocks* the performance is increased by 0.53% while tripling the number of parameters. More blocks means more time ($T = h * n_{blocks}$) for the Hamiltonian to modify and extract features from the network and that may be why the performance increases.
The three ways to increase the size of the network are interesting and have to be considered, but the more profitable one (after changing the kernel size from 1 to 3) seems to be increasing the number of units. In parallel, one could fight overfitting using more dataset extensions methods.

5.2 Comparison with other networks

In the sum-up table 4.7, the performance of all the networks is compared.
The H-DNN from Chang et al. [2] provides a better result than what is presented in this report. The reason why is unclear. But we were able to improve our implementation of their network and to find interesting results, despite of all of them being weaker than the original ones. Maybe I missed some information in their paper, or the information was missing. Another option is that they used 2D or 3D grid searches where I used only 1D ones. Also, they may have done thinner grid searches than me (I mainly looked at different orders of magnitudes).
When compared to the simple *AlexNet* [4], the Hamiltonians use much less parameters while reaching better performance. The deepness of the Hamiltonians compared to the *AlexNet* [4] may be the explanation.
Compared to the residual networks (*ResNet* [5]), the Hamiltonian achieve comparable performance but systematically lower. There is no obvious explanation. However, the main force of the Hamiltonians is their proven stability, which *ResNet* [5] has not.
Compared to the current best state of the art networks, Hamiltonians Networks perform much poorly. Nevertheless, the actual ones use at least 500 times more parameters. Moreover, the innovations they introduce are not in the architecture, like Hamiltonians do. Instead they provide more optimised way to train a network (new batch-splitting from *GPipe* [7] and new loss from *SAM* [10]), or methods to reuse networks trained on much bigger datasets to smaller ones (*Bit-L* [9]). So, the comparison of Hamiltonians with them has to be done carefully: they provide incredible performance, but they use many parameters and work on a different aspect of DL than H-DNN do.
A method that works on the same aspect of DL as H-DNN is batch normalisation. It has been proposed by Ioffe et Szegedy in 2015 [13]. Among others goals, it leads to fewer gradient issues by normalising (re-centring and re-scaling) the input of the layers at each batch. Despite being widely use with good success there is still debate about why does it work (three different explanations: [13], [14] & [15]). On the opposite, H-DNN has strong mathematical proofs which motivates its exploration.

6. Conclusion

This report aimed to implement Hamiltonian Deep Neural Networks for the Image Classification task. This architecture ensures that the network will not face exploding or vanishing gradient [1]. The provided solution is based on what Chang et al. [2] introduced: the network is divided into various *units* composed of a certain number of Hamiltonian *blocks*, a data map pooling and a channel padding. Before the first *unit* a convolutional layer is used and after the last *unit* a fully connected layer classifies the images. Some additional layers have been introduced, like dropout and batch normalisation. The network has also been tested with new Hamiltonian architectures (J1 and J2, from Galimberti et al. [3]).

The study of Hamiltonian Deep Neural Network led to the following conclusions:

- These networks can solve the Image Classification task.
- They can be easily adapted to different datasets: more labels (CIFAR-100 instead of CIFAR-10), less data (10% of CIFAR-10) or even different image shape (STL-10 instead of CIFAR-10).
- Discretisation timestep, smoothing regularisation weight and learning rate are three crucial parameters to ensure correct and efficient learning when using Hamiltonians Neural Networks and thus have to be selected carefully.
- J1 (with ReLU activation) is the best-suited architecture for Image Classification (it achieves **92.27% with less than one million of parameters on CIFAR-10**).
- ReLU outperforms Tanh activation for the three architectures.
- From what been tested, the most efficient regularisation loss multiplies it by h and includes the loss based on $\boldsymbol{b}$.
- The additional dropout and batch normalisation layers have a good influence on the networks.
- The initialisation of the layers has to be wisely done, Xavier or Xavier-like are efficient for this task.
- Adam optimiser could be used instead of SGD.
- The most efficient way to enlarge the network would be to increase the number of units.

Despite these exciting findings, the Hamiltonian Networks perform similarly to standard methods and are entirely outperformed by the actual state of the art networks (that use much bigger networks) ([7], [9] & [10]). Nevertheless, they are one of the few *proven* methods to enforce that the gradient neither vanishes nor explodes.

Bibliography

[1] HABER, Eldad et RUTHOTTO, Lars. Stable architectures for deep neural networks. Inverse Problems, 2017, vol. 34, no 1, p. 014004.

[2] CHANG, Bo, MENG, Lili, HABER, Eldad, et al. Reversible architectures for arbitrarily deep residual neural networks. In : Proceedings of the AAAI Conference on Artificial Intelligence. 2018.

[3] GALIMBERTI, Clara L., XU, Liang, et TRECATE, Giancarlo Ferrari. A unified framework for Hamiltonian deep neural networks. arXiv preprint arXiv:2104.13166, 2021.

[4] KRIZHEVSKY, Alex, SUTSKEVER, Ilya, et HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 2012, vol. 25, p. 1097-1105.

[5] HE, Kaiming, ZHANG, Xiangyu, REN, Shaoqing, et al. Deep residual learning for image recognition. In : Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 770-778.

[6] IDELBAYEV, Yerlan Idelbayev18a, Proper ResNet Implementation for CIFAR10/CIFAR100 in PyTorch, github: `https://github.com/akamaster/pytorch_resnet_cifar10`, Accessed in April 2021

[7] HUANG, Yanping, CHENG, Youlong, BAPNA, Ankur, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 2019, vol. 32, p. 103-112.

[8] REAL, Esteban, AGGARWAL, Alok, HUANG, Yanping, et al. Regularized evolution for image classifier architecture search. In : Proceedings of the aaai conference on artificial intelligence. 2019. p. 4780-4789.

[9] KOLESNIKOV, Alexander, BEYER, Lucas, ZHAI, Xiaohua, et al. Big transfer (bit): General visual representation learning. arXiv preprint arXiv:1912.11370, 2019, vol. 6, no 2, p. 8.

[10] FORET, Pierre, KLEINER, Ariel, MOBAHI, Hossein, et al. Sharpness-Aware Minimization for Efficiently Improving Generalization. arXiv preprint arXiv:2010.01412, 2020.

[11] XIE, Qizhe, LUONG, Minh-Thang, HOVY, Eduard, et al. Self-training with noisy student improves imagenet classification. In : Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020. p. 10687-10698.

[12] GLOROT, Xavier et BENGIO, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In : Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2010. p. 249-256.

[13] IOFFE, Sergey et SZEGEDY, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In : International conference on machine learning. PMLR, 2015. p. 448-456.

[14] SANTURKAR, Shibani, TSIPRAS, Dimitris, ILYAS, Andrew, et al. How does batch normalization help optimization?. arXiv preprint arXiv:1805.11604, 2018.

[15] KOHLER, Jonas, DANESHMAND, Hadi, LUCCHI, Aurelien, et al. Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization. In : The 22nd International Conference on Artificial Intelligence and Statistics. PMLR, 2019. p. 806-815.

[16] KINGMA, Diederik P. et BA, Jimmy. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

A. Appendices

A.1 Framework utilisation

A.1.1 Minimal code example

The code snippet A.1, shows a minimal example of **run.py** that trains and tests a network. The user has to define the dataset used and the three structures that define the whole problem (or load preexisting structures and modify them, as in the example). Then, by calling the *train_ and_ test* function, the model will be trained and tested, and the performance and loss returned (final and along epochs). Note that due to space constraints, the output is not saved in the example. For the same reason, the GPU or CPU information (*device* and *kwargs*) are not defined.
Note that in case of a *NaN* value in the output or of very high loss ($> 10^{15}$) the network stops training and returns accuracy of 0 and loss of 1 for the final loss and performance as well as for the rest of the epochs.

```
import torch
import params as p
from train_and_test import train_and_test

DATASET = "CIFAR10"

#Load a pre-existing network
learn_params = p.best_paper_learn_params
net_params = p.best_paper_net_params

#Modify some parameters
learn_params.batch_size = 42
net_params.ham_params.h = 1e-2

_, _, _, _, _, _,_,_ = train_and_test(DATASET, net_params,
                              learn_params, device, kwargs)
```

Figure A.1: Minimal code example.

A.1.2 Code organisation

The code is divided in various *Python* files. The interconnections between the different files are plotted in figure A.2. In a nutshell, **run.py** or **run_grid.py** will call **train_and_test.py** which will train and test a network defined by the three *dataclasses* received as parameters from **run.py** or **run_grid.py**. To do so, this file will call others and finally return to the file who called it the performance achieved by the network.
Here is a list of the main role for each file:

- **params.py**: define the parameters *dataclasses* that define the network and how it will be trained;
- **run.py**: define and run a network using the parameters structures by calling **train_and_test.py**, the obtained performance is saved in a folder;
- **run_grid.py**: run a grid search over one or two parameters, calls **train_and_test.py** for each combination of parameters, the obtained performance is saved in a folder;
- **train_and_test.py**: create, train and evaluate a network based on the parameters received;
- **datasets.py**: generate the datasets (with pre-processing steps);
- **networks.py**: define the network class and sub-classes;
- **hamiltonians.py**: define the different Hamiltonians classes;
- **regularisation.py**: implement the smoothing regularisation function;
- **utils.py**: define utilities function for the whole project (ex.: initialise a layer with a certain methods);
- **plot.py**: plot the results obtained by **run.py** or **run_grid.py**;
- **plot_compare.py**: plot a comparison between various results obtained by **run.py**.

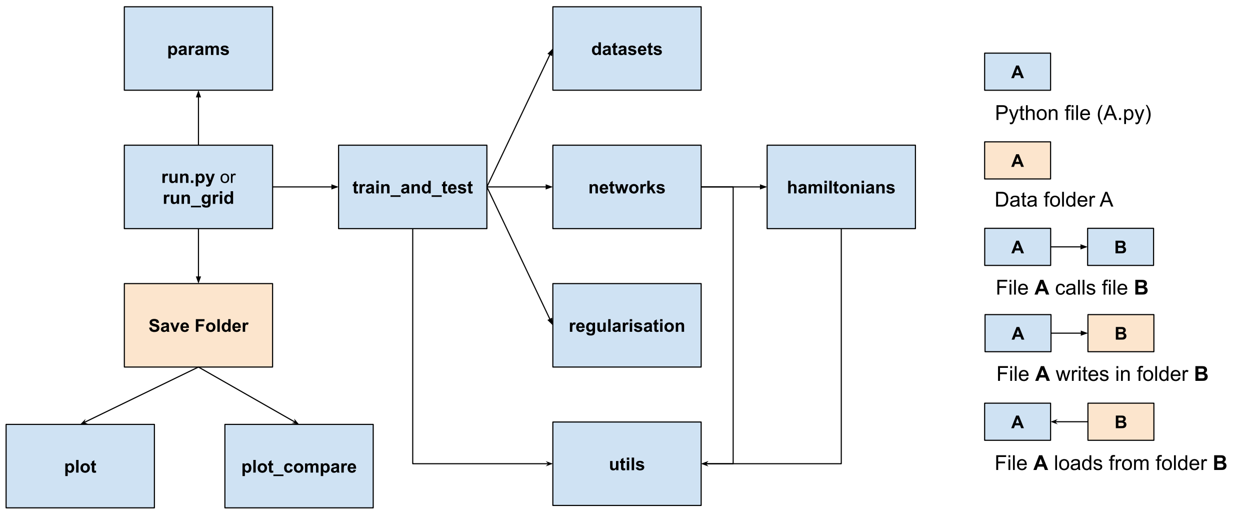


Figure A.2: Organisation of the code.

A.2 Implementation details

A.2.1 Learning process

A.2.1.1 Classical Weight Regularisation (wd)

Note that the classic weight regularisation, weight decay (enforcing that the weights do not have too big values), is done directly inside the optimiser. That's why, in the pseudo-code 1, the optimiser receives as input the weight regularisation weight, *wd*.

A.2.1.2 Weight Smoothness Regularisation (alpha, b_in_reg & h_div)

As pointed out by Haber and Ruthotto [1], H-DNN need weight smoothness regularisation (i.e. ensure that matrix $\boldsymbol{K}$ do not change too fast over time, neither vector $\boldsymbol{b}$). Here, it has been implemented in a way to let the user choose between the different possibilities (2.5, 2.17 and 2.22).
Let us define R as the weight smoothness regularisation loss that will be added to the original loss, it can be defined as:

$$R = \begin{cases} alpha * (R(\boldsymbol{K}) + R(\boldsymbol{b})) & b\_in\_reg \text{ is True} \\ alpha * R(\boldsymbol{K}) & b\_in\_reg \text{ is False} \end{cases} \tag{A.1}$$

Where b_in_reg and $alpha$ are parameters from the *LearningParameters* structure. The first one enables or disables the weight smoothing of $\boldsymbol{b}$. The second one is simply a weight w.r.t. to the original loss. $R(\boldsymbol{K})$ and $R(\boldsymbol{b})$ are defined as:

$$R(\boldsymbol{K}) = \tilde{h} \sum_{k=1}^{M} \sum_{j=1}^{N-1} \|\boldsymbol{K}_j - \boldsymbol{K}_{j-1}\|_F^2 \text{ and } R(\boldsymbol{b}) = \tilde{h} \sum_{k=1}^{M} \sum_{j=1}^{N-1} \|\boldsymbol{b}_j - \boldsymbol{b}_{j-1}\|^2 \tag{A.2}$$

F refers to the Frobenius norm, M to the number of *units* and N to the number of Hamiltonian *blocks* in each *unit*. The value $\tilde{h}$ depends on another parameters from *LearningParameters*: h_div which defines if h has to multiply or divide the norm :

$$\tilde{h} = \begin{cases} \frac{1}{h} & h\_div \text{ is True} \\ h & h\_div \text{ is False} \end{cases} \tag{A.3}$$

Clara et al. [3] suggest to use multiplication (see equation 2.22), where as Haber and Ruthotto [1] and Chang et al. [2] use division (see equations 2.5 and 2.17).

A.2.1.3 Optimiser (optimiser)

The first option is "SGD", which refers to the well known Stochastic Gradient Descent. Its hyperparameter is the "momentum".
The second option is "Adam" and refers to Adam optimiser [16] (ADAptative Moment estimation). This optimiser updates the weights as follows:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{A.4}$$

Where w_t are the parameters of the network at instant t, η is the learning rate and ϵ is a small number for numerical stability. $\hat{m}_t$ and $\hat{v}_t$ are computed as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \end{aligned} \tag{A.5}$$

β_1 and β_2 are two hyperparameters. g_t is the gradient at timestep t.
When looking more precisely to the equations one may notice that m_t is the first degree moment of the gradient. It is estimated through a moving average whose decay is determined by β_1. The bias of this estimator is corrected and results in $\hat{m}_t$. Same idea applies to the second degree moment v_t, whose decay is determined by β_2.
The weights are modified with respect to these two moments. $\hat{v}_t$ divides the learning rate, thus it is adaptable (compared to SGD). Moreover, it can be different for all the parameters. η bounds the maximum step size that Adam can do.
These properties allow the optimiser to deal with more complicated objective function.

A.2.2 Network structure

A.2.2.1 Number of channels (nb_features)

The evolution of the number of channels along the network can be defined using the $nb\_features$ list. Note that the number of channels can never diminish. The first value of the list has to be the number of layer of the input data. Then, each value will indicate the desired number of channels at the input of the corresponding *unit*. Zero-padding layers will be used to complete the missing channels.

A.2.2.2 Initialisation (init)

The convolutional layer and the fully connected layer(s) can be initialised in four different ways:

- $init = "Zeros"$: weights are set to 0;
- $init = "Ones"$: weights are set to 1;
- $init = "Normal"$: weights are generated from a normal distribution (mean=0, std=0.01);
- $init = "Xavier"$: weights are generated from a Xavier distributon (PyTorch documentation).

A.2.2.3 Additional fully connected layer (second_final_FC)

An additional FC layer can be added at the end of the network if $second\_final\_FC$ is not *None*. In this case, the value of $second\_final\_FC$ will define the output dimension of the first FC layer and the input dimension of the second one.

A.2.2.4 Padding (both_border_pad)

In the padding layer, channels full of zeros will be added to the existing channels. If *both_border_pad* is *False*, the new channels (full of zeros) will be added at the end of the already existing channels. On the opposite, if it is *True*, half of the new channels will be added before the original channels and the other half after.

A.2.2.5 Dropout (dropout)

The dropout will randomly put to zero some neurons and re-scale the others to maintain consistency (PyTorch documentation). It can be disabled by setting *dropout* to *None*. Else, it expects to receive a list of two elements, first the dropout probability of the first dropout layer (before the first *unit*) and second the dropout probability of the second one (before the classification layer).

A.2.3 Hamiltonians

A.2.3.1 Initialisation (init)

K and **b** can be initialised in four different ways:

- $init = "Zeros"$: weights/biases are set to 0;
- $init = "Ones"$: weights/biases are set to 1;
- $init = "Normal"$: weights/biases are generated from a normal distribution (mean=0, std=0.01);
- $init = "Xavier"$: weights/biases are generated from a distribution (mean=0, std=σ). Defining N as the number of input channels of the Hamiltonian Layer and ks as the kernel size of the convolutional operator $\mathbf{K}$, σ is:
$$\sigma = \sqrt{\frac{2}{N * ks * ks}} \tag{A.6}$$
Please note that the latter initialisation is not a standard Xavier initialisation. This formula comes from the *GitHub* of Chang et al. [2] but for another project (GitHub link). The wrong name is used to have the same initialisation options in *HamiltonianParameters* and *NetworkParameters*. Nevertheless, this initialisation has similarities with the Xavier one, as both depend on the square root of the inverse of the input dimension.

A.3 Laboratory Computer

All the networks were trained on the laboratory computer wirelessly. The utilisation of the workstation was not complicated and pleasant. In my point of view, there were only two issues (but I never asked help for them so maybe they are easy to fix).
First, I had to let the terminal that launched the code on my computer open, otherwise the code stopped. Thus, I could not shut down my computer and I had to stay connected to the EPFL VPN all along.
Second, and directly related to the first issue, the EPFL VPN in my computer disconnected once per day. It may be a security from EPFL. Thus, I was not able to launch grid searches that last for more than 24 hours. A standard network from this report takes around 2 hours to train and thus I could maximum train 12 networks per day. Which means that for a 2D grid search I could maximum test 3-4 values for each parameter, which is very low. That is the reason why in this report there is only 1D grid searches.

A.4 Chang et al. Network Implementation Assumptions

Table 4.1 defines the parameters of our implementation of the Chang et al. network [2]. In it, some parameters are written in italic. These parameters had to be assumed due to lack of information and thus may not have the exact value that Chang et al. used.
For some of them, the assumptions made were natural. It has been assumed that the optimiser used is SGD since they mentioned the "momentum". For the convolutional layer, a classic kernel of 3x3 was selected and then the stride and padding were deduced to not alter the image dimension. It was also assumed that they are using Cross Entropy Loss, as it is the most classic choice in this case.
However, for other parameters, the assumptions are weaker and may have a strong influence on the performance. First, the initialisation was assumed to be Xavier for the convolutional and fully connected layers. For **K** and **b**, the initialisation method is presented in Appendix A.2.2 and is inspired by code from their *GitHub* but for another project. The kernel size (ks) for **K** has been set to 3 and the discretisation time (h) to 0.05 (a grid search has been used to determine this value). These two last parameters have a strong influence on the performance and may explain the difference between the results obtained by this network in this report and their results.
Another element which may also impact the performance is their definition of "epoch". In this report, a *training step* is defined as one model update, the *batch size* is the number of data points presented to the network to update it and an *epoch* is achieved when the network has seen once all the data. These definitions concur to what can be found in my previous classes and on Internet (*machinelearningmastery.com*, *towardsdatascience.com*, *radiopaedia.org* and *quora.com*). However, when using these definitions what they give us, strange values appear. For example when dealing with STL-10 they say that they use a batch size of *128* and that they do 20'000 *training steps*. Thus, each epoch results in 40 training steps (= 5000/128 (5000 training images in STL-10)). So, in total there would be 500 epochs ($20'000/40$). But, they propose to reduce the learning rate at 60, 80 and 100 epochs. In the same fashion, for CIFAR-10, the last learning rate update happens in the very last epoch. Both seem strange and so there may be a differences in the definition of an *epoch* between their paper and this report, which may explain some of the performance difference.

A.5 Additional plots

A.5.1 Hamiltonian Network - J1 - ReLU

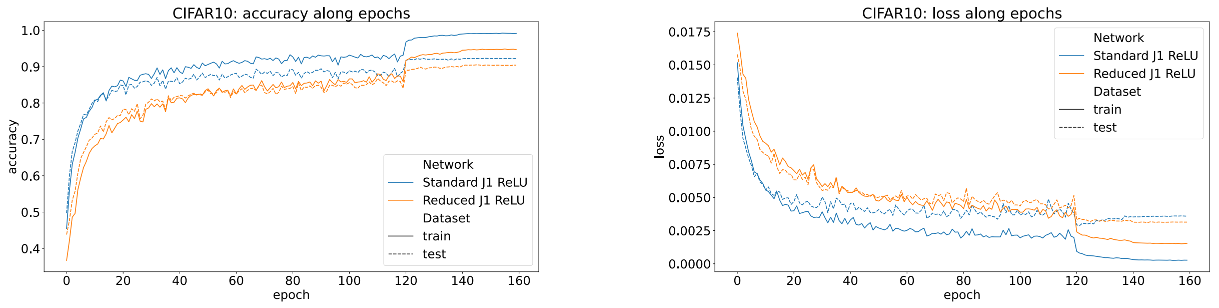


Figure A.3: Evolution of the loss and accuracy along epochs of J1-ReLU Hamiltonian based networks on CIFAR-10. In orange a reduced version of the blue network (half number of parameters).

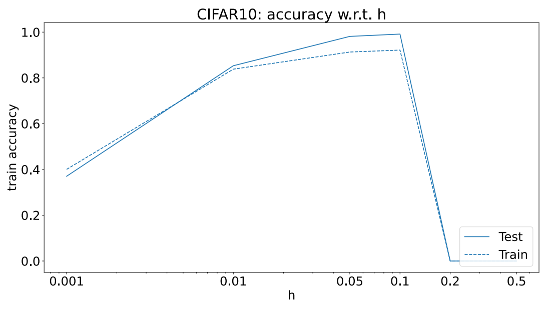


Figure A.4: Influence of h on the performance of the J1-ReLU network on CIFAR-10.

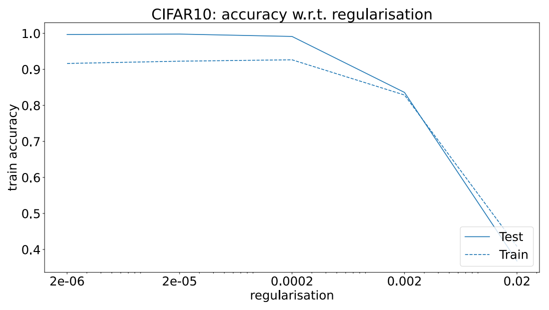


Figure A.5: Influence of the regularisation's weights (smoothing $(alpha)$ and decay (wd)) on the performance of the J1-ReLU network on CIFAR-10.

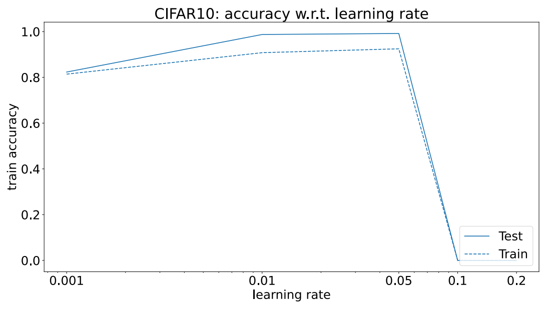


Figure A.6: Influence of the learning rate on the performance of the J1-ReLU network on CIFAR-10.

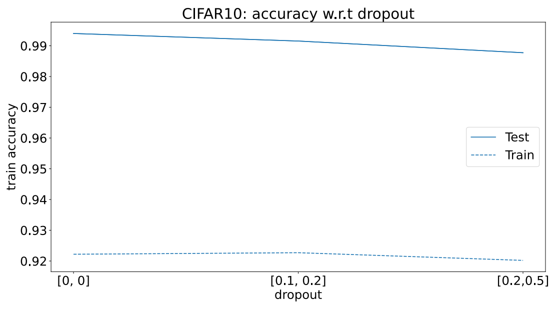


Figure A.7: Influence of the dropout on the performance of the J1-ReLU network on CIFAR-10.

A.5.2 Hamiltonian Network - J1 - Tanh

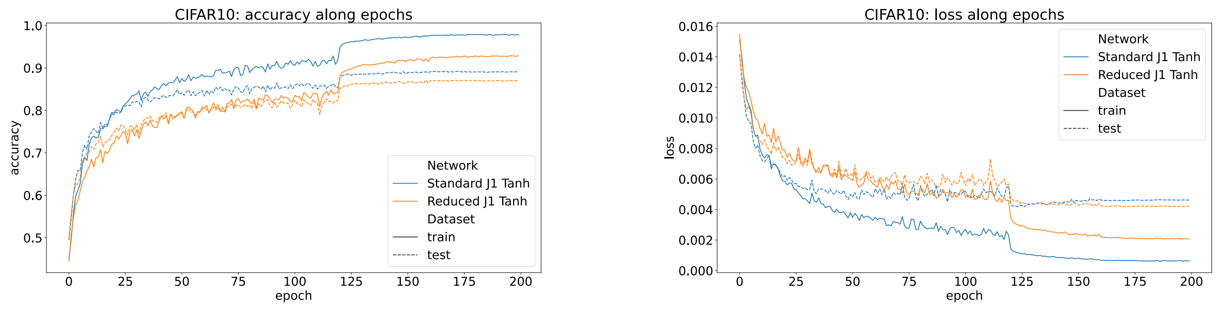


Figure A.8: Evolution of the loss and accuracy along epochs of J1-Tanh Hamiltonian based networks on CIFAR-10. In orange a reduced version of the blue network (half number of parameters).

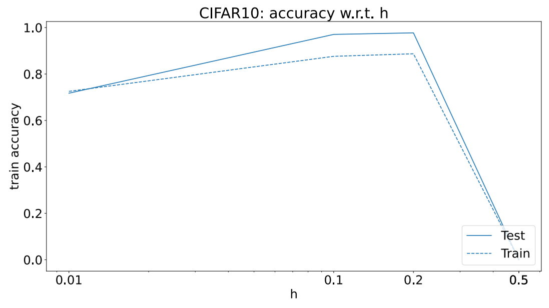


Figure A.9: Influence of h on the performance of the J1-Tanh network on the CIFAR-10 dataset.

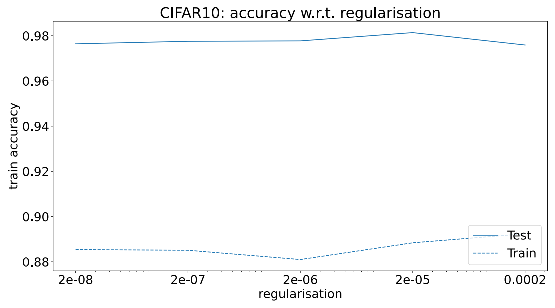


Figure A.10: Influence of the regularisation's weights (smoothing ($alpha$) and weight decay (wd)) on the performance of the J1-Tanh network on the CIFAR-10 dataset.

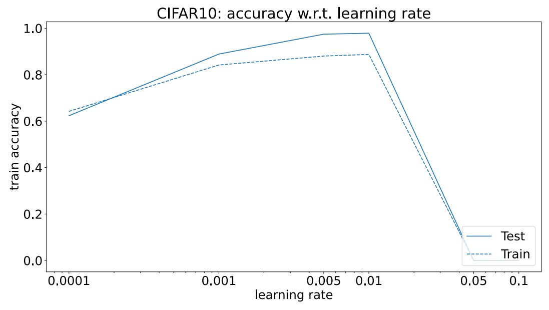


Figure A.11: Influence of the learning rate on the performance of the J1-Tanh network on the CIFAR-10 dataset.

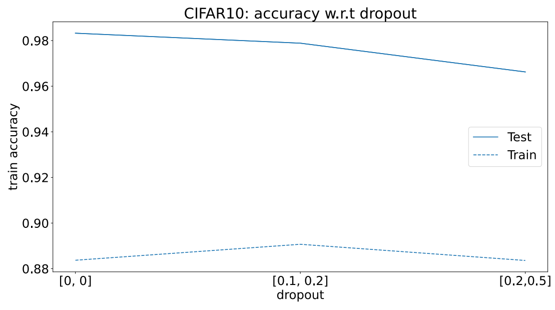
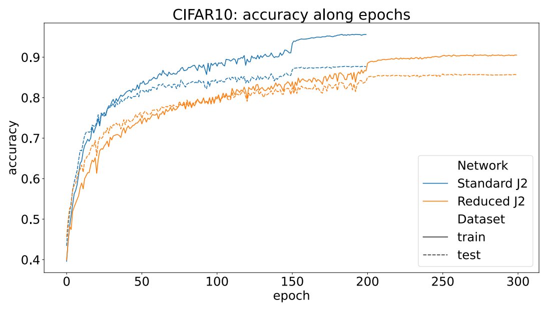


Figure A.12: Influence of the dropout on the performance of the J1-Tanh network on the CIFAR-10 dataset.

A.5.3 Hamiltonian Network - J2



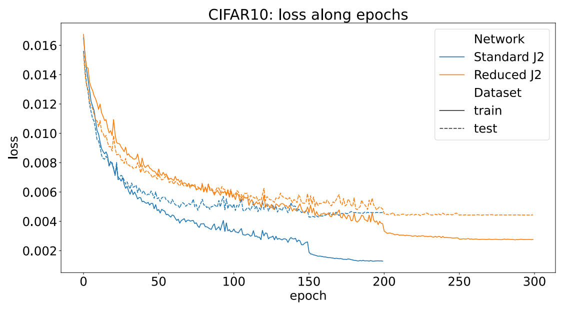


Figure A.13: Evolution of the loss and accuracy along epochs of J2 Hamiltonian based networks on CIFAR-10. In orange a reduced version of the blue network (half number of parameters).

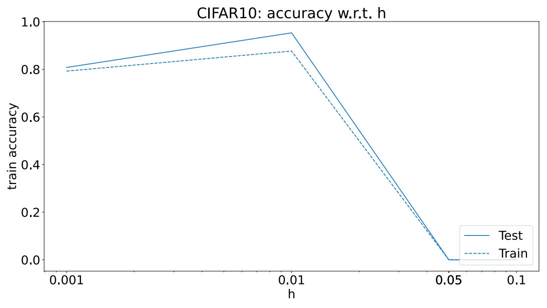


Figure A.14: Influence of h on the performance of the J2 network on the CIFAR-10 dataset.

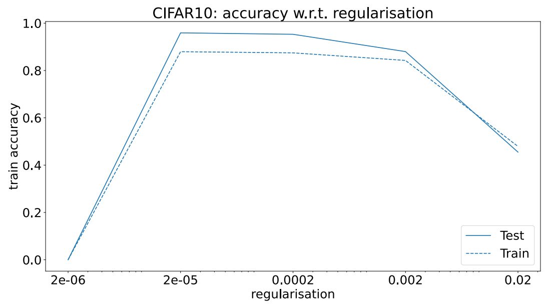


Figure A.15: Influence of the regularisation's factors (smoothing ($alpha$) and weight decay (wd)) on the performance of the J2 network on the CIFAR-10 dataset.

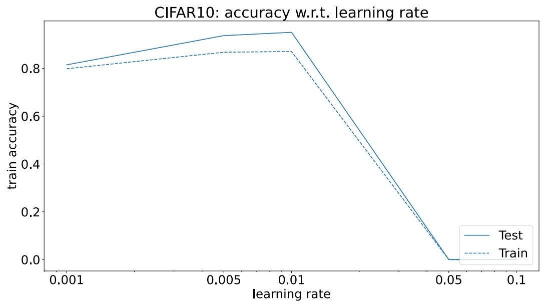


Figure A.16: Influence of the learning rate on the performance of the J2 network on the CIFAR-10 dataset.

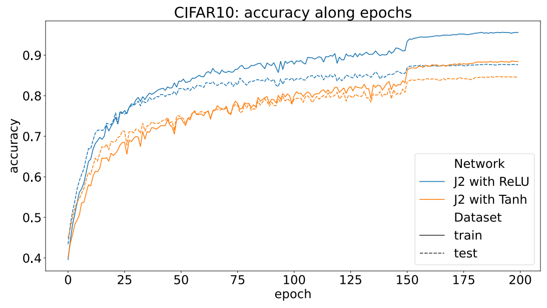


Figure A.17: Accuracy along epochs for J2 Hamiltonian networks on CIFAR-10. In blue, using ReLU activation. In orange, using Tanh activation.