Binary classification as a phase separation process - a short tutorial to the implemented model

Rafael Monteiro

2020-09-17

Contents

Chapter 1

Introduction

This is a short companion tutorial to the github (Monteiro, 2020b). We give here a glimpse of the Phase Separation Binary Classifier (in short, PSBC), proposed in the article **Binary Classification as a Phase Separation Process**, by Rafael Monteiro; a preprint is available on arXiv.

An implementation of the model can be found in the module **binary_phase_separation.py**. Part of this tutorial can also be found in README.pdf.

...but, before we start...

- The main module for this project is **binary_phase_separation.py**, that you will find in this Github page (Monteiro, 2020b).
- If you are looking for the dataset and trained models, see the (Monteiro, 2020a) at Zenodo.
- If you want to reproduce this notebook you should download the file PSBC_Examples.tar.gz in the aforementioned data repository, and also the tarball jupyter-notebooks in (Monteiro, 2020b).[1]
- To download data, I highly recommend the use of the script download_PSBC.sh, for a few reasons: (i) it downloads the data in a safe way (using wget); (ii) it automatically checks the MD5 of each file. We would like to highlight that you can also use the same script to download only the trained examples and only the jupyter-notebooks for this project. In order to use it, please read first this README.pdf guide for further explanation.
- This website is based on this jupyter-notebook. The full code to the pictures seen here are posted there. For that reason, I shall refrain from posting the whole oputput of some cases, marking then with a symbol >>>.
- We will summarize some of the outputs, adding a "..." to them. If you would like to see the whole output, please see Notebook_PSBC_example.ipynb.

As we said, this discussion is short but we want to offer a bit more than just a manual, so we will point out references along the way.

[1] This tarball contains all the 3 notebooks, and also the two modules used.

Chapter 2

Nonlinear diffusion equations: a numerical example

To begin with, I will introduce the model, giving some examples of its use. Let's first import some libraries

```
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.preprocessing import MinMaxScaler
import matplotlib.gridspec as gridspec
...
```

As discussed in Section 1.1 in the paper, the nonlinear diffusion processes are in the backdrop of this model. The heart of the model is the Allen-Cahn equation (Fife, 1979),(Aronson and Weinberger, 1978),(Allen and Cahn, 1979), a well-known equation in the field of pattern formation. Just to show how part of the PSBC's implementation can be used to evolve an initial boundary value problem with Neumann boundary conditions: we shall take

$$u_0(x) = \frac{1 - \sin(\pi(2x-1))}{2} \tag{2.1}$$

as an initial condition to the Allen-Cahn equation

$$\partial_t u(x,t) = \varepsilon^2 \partial_x^2 u(x,t) + u(x,t)(1-u(x,t))(u(x,t) - \alpha(x)). \tag{2.2}$$

The parameter $\alpha(\cdot)$ embodies medium heterogeneity. In the case shown here, we choose $\alpha(x) = -2$, when $x < 0.5$, and $\alpha(x) = 2$, when $x \geq 0.5$.

Parameters to the model are assigned below:

```
N = 20
x = np.linspace(0, 1, N, endpoint = True)
V_0 = 1/2 - 1/2 * np.reshape(np.sin(np.pi * (2 * x - 1)) , (-1,1))
prop = Propagate()
dt, eps, Nx, Nt = 0.1, .3,  N, 400
dx, ptt_cardnlty, weigths_k_sharing = x[1]-x[0], Nx, Nt
```

Then we initialize parameters

```
init = Initialize_parameters()
param = init.dictionary(N, eps, dt, dx, Nt, ptt_cardnlty, weigths_k_sharing)
```

If you read the paper you remember that trainable weights are the coefficients of this PDE. Since the model randomly initialize these coefficients, we will have to readjust them to the value we want. That's what we do in the next part of the code.

```
for i in range(param["Nt"]):
        param["alpha_x_t"][:,i] = -2 * (x < .5) + 2 * (x >= .5)
```

which we now run, using the numerical scheme (1.7a) in the paper. Recall that this step is the same as doing a forward propagation in a feed forward network: that's the reason why you see the method "prop.forward" in the code below.

```
flow, waterfall, time = prop.forward(V_0, param, waterfall_save = True , Flow_save = True)
time = np.arange(Nt + 1)
X, Y = np.meshgrid(x, time)
flow = np.squeeze(flow, axis = 1)
```

When we plot the evolution of $u_0(\cdot)$ through the Allen-Cahn equation as a surface, we get the plot below (code in Notebook_examples.ipynb).

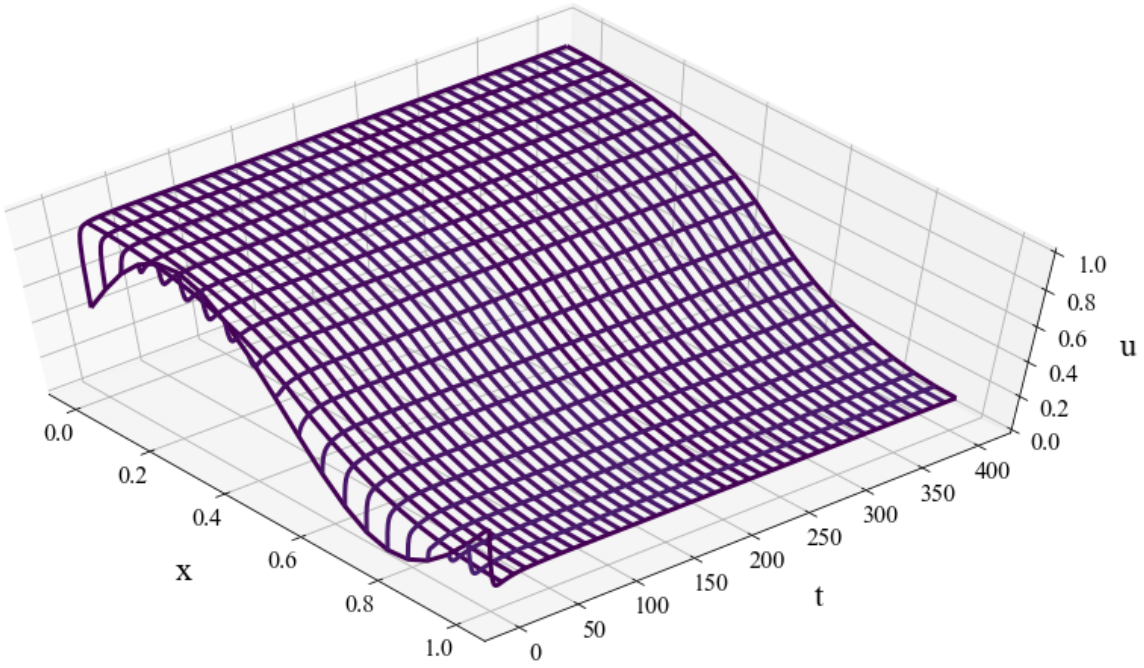


Figure 2.1: The initial condition $u_0(\cdot)$ shown in Equation (2.1), as it evolves according to the Allen-Cahn equation (2.2).

One of the first motivations to this project can be found in the interesting paper (Angenent et al., 1987), where they have shown that several nontrivial layered patterns^{Stationary solutions to equation (2.2) that, roughly speaking, gets "concentrated" around values 0 and 1, displaying layers in between these values.} can be found if $\alpha(\cdot)$ is non-homogeneous in space. There is an extensive discussion about why this is interesting, and we refer the reader to Section 1.1 in the paper.

> **Remark:** You can read more about pattern formation in the book (Nishiura, 2002) (Chapter 4.2 deals with the Allen-Cahn model), and also in the very nice article Arnd Scheel wrote to the Princeton Companion to Applied Mathematics, Section IV. 27 (Dennis et al., 2015).

2.1 Propagation with randomly generated coefficients

Next, we would like to evolve and plot the evolution of several different initial conditions in the interval [0,1] (code in Notebook_examples.ipynb). The model that we use is, initially, a discretization of (2.2), with $\varepsilon = 0$, which then becomes an ODE:

$$U^{[n+1]} = U^{[n]} + \Delta_t^u f(U^{[n]}, \alpha^{[n]}), \tag{2.3}$$

where $f(U^{[n]}, \alpha^{[n]}) := U^{[n]}(1 - U^{[n]})(U^{[n]} - \alpha^{[n]})$.

It is good to have in mind that the coefficients in the above ODE will play the role of trainable weights in Machine Learning: we will "adjust" the coefficients in $\alpha(\cdot)$ in order to achieve a certain final, target end state.

As mentioned earlier, there is a clear correspondence between the initial value (ODE/PDE) problem and forward propagation and, consequently, the stability of (2.3) has to be considered. The discretization it presents is known as (explicit) Euler method, which is known to be (linearly) unstable in many cases. A good part of the paper was devoted to showing that there is some kind of *nonlinear stabilization* mechanism that prevents solutions from blowing up, a condition referred to as *Invariant Region Enforcing Condition*, which establishes a critical threshold for the size of Δ_t^u, beyond which solutions can blow up. This is discussed at lenght in the paper.

To get this critical value for Δ_t^u it is necessary to quantify

$$\max_{1 \le k \le \mathrm{N_t}} \max\{1, |\alpha^{[k]}|\} < +\infty, \tag{2.4}$$

based on which we adjust the parameter Δ_t^u accordingly, in a nontrivial way.^{Because, of course, $\Delta_t^u = 0$ also does the job, but does not deliver what we want.} In result, the evolution of $U^{[\cdot]}$ does not end up in a floating point overflow (in other words, a blow up in ℓ^∞ norm).

We set up a little experiment, where in some we obey the *Invariant Region Enforcing Condition*, and in some we don't. We take several initial condition on the interval $[0, 1]$ and evolving them according to (2.3). Parameters are as follows:

```
N = 1
init = Initialize_parameters()
prop = Propagate()
dt_vec = np.array([.1,.3,.57,1.5,3,4])
dt, eps, Nx, Nt, dx = .1, 0, N, 20, 1
ptt_cardnlty, weights_k_sharing = Nx, Nt
```

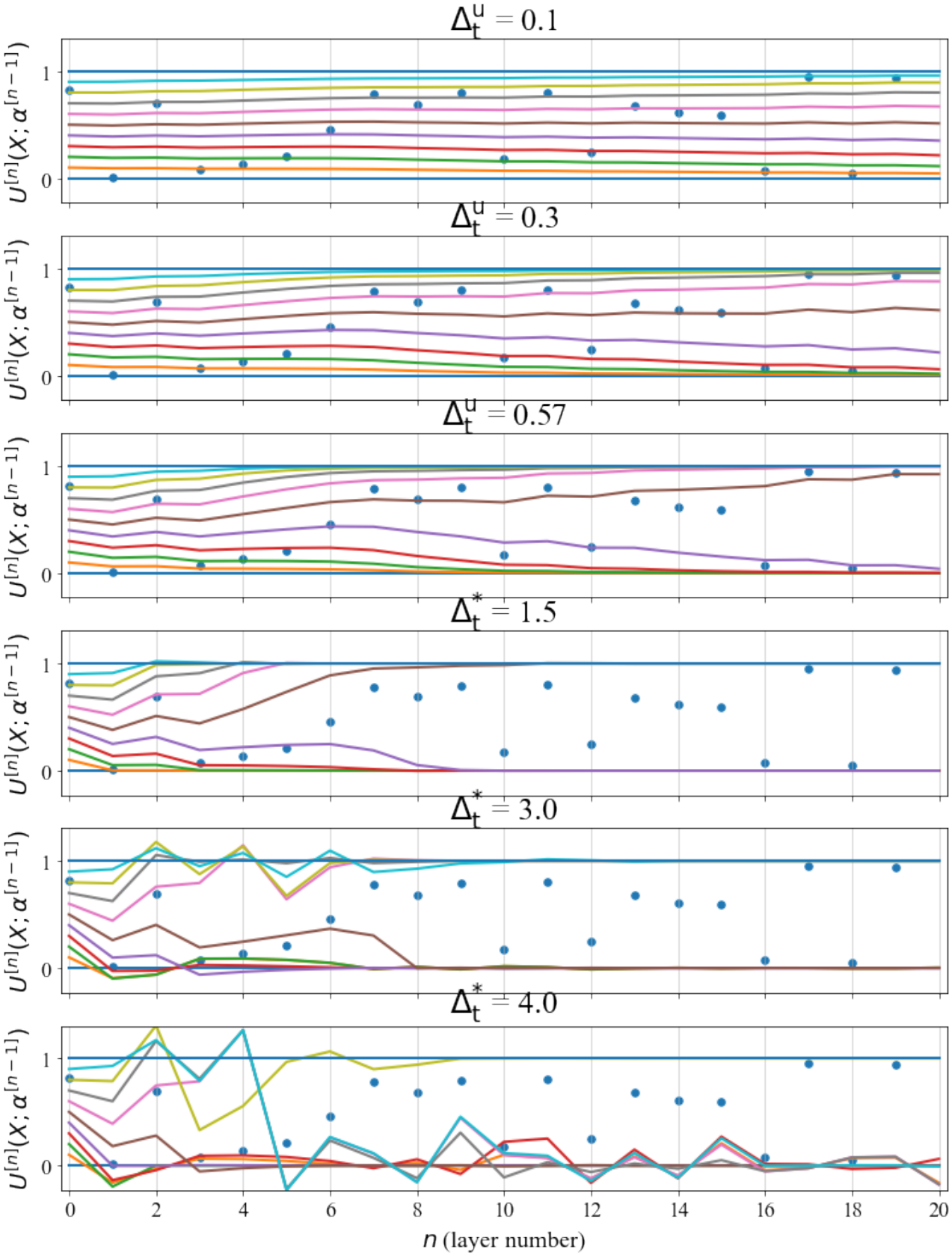
As discussed in the Appendix A in the paper, the PSBC randomly initializes trainable weights coefficients as realizations of a Normal random variable with average 0.5 and variance 0.1. We set them as uniform random variables in the interval [0,1], which implies that the left hand side of (2.4) is bounded by 1.

```
param = init.dictionary(N, eps, dt, dx, Nt, ptt_cardnlty, weights_k_sharing)

for i in range(param["Nt"]): param["alpha_x_t"][:,i] = np.random.uniform(0,1)

n_points = 10
V_0 = np.reshape(1/n_points * np.arange(0, n_points + 1), (1, -1))
flow, waterfall, time = prop.forward(V_0, param, waterfall_save = True , Flow_save = True)
```

We obtain the following figure.



The reasoning behind the existence of critical values of Δ_t^u under which the solution is "well behaved" (that

is, the solution is always bounded) goes back to the idea of *Invariant regions*, exploited extensively in PDEs: we refer the reader to Chapter 14 in (Smoller, 1994); if you want to see how it applies in the discrete setting, especially in finite-difference schemes for reaction diffusion models, see (Hoff, 1978) and the Appendix C of the paper).

Chapter 3

Applying the PSBC model to some toy problems

We shall present the model in a simple toy problem, for illustrative purposes. As pointed out in the paper, we have to use a somewhat "bigger"" model, of the form

$$\begin{aligned} U^{[n+1]} &= U^{[n]} + \Delta_t^u \, f(U^{[n]}, \alpha^{[n]}), \\ P^{[n+1]} &= P^{[n]} + \Delta_t^p \, f(P^{[n]}, \beta^{[n]}), \end{aligned} \tag{3.1}$$

where $f(u, w) := u(1-u)(u-w)$. Now, given a certain cost function that measures accuracy of the model (namely, how well it predicts in some examples) we have to train the model with respect to both variables $\alpha^{[\cdot]}$ and $\beta^{[\cdot]}$. What we see in (3.1) is part of what we call Phase Separation Binary Classifier (PSBC). For now, we shall see how it behaves in the 1D model; in Section 4, after many variations over this equation, we shall apply the model to the MNIST dataset.

3.1 The 1D Rectangular box problem

We shall work with a simple 1D model (the rectangular box problem), with the following labeling method:

$$Y = Y(X) = \begin{cases} 1, & \text{if} \quad X \geq \gamma; \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

```
folder = "Statistics/MNIST/"
with open(folder + "parameters_MNIST_Neumann.p", 'rb') as fp: data = pickle.load(fp)
```

```
### GENERATE DATA
gamma, N_data = .2, 2000
X = np.reshape(np.random.uniform(0, 1, N_data),(1, -1))
Y = np.array(X >= gamma, np.int, ndmin = 2)

### SPLIT DATA FOR CROSS VALIDATION
A, B, C, D = train_test_split(X.T, Y.T, test_size = 0.2)
#### We shall save one individual per column. We need to change that upon reading the csv later on
X_train, X_test, Y_train, Y_test = A.T, B.T, C.T, D.T
```

In this model, the data has to satisfy features dimension X number of elements in the sample

```
np.shape(X_train)
```

>>>

```
  (1, 1600)
```

Things go more or less as before: we define the model's parameters,

```
learning_rate = (.1,.08,.93)
patience = float("inf")
sigma = .1
drop_SGD = 0.95  # See  docstring of class "Binary_phase_separation" for further information
epochs, dt, dx, eps, Nx, Nt = 600, .1, 1, 0, 1, 20
weights_k_sharing = Nt
ptt_cardnlty = 1
batch_size = None
subordinate, save_parameter_hist, orthodox_dt, with_phase = True, True, True, True
```

and initialize the model

```
Init = Initialize_parameters()

data = Init.dictionary(Nx, eps, dt, dx, Nt, ptt_cardnlty, weights_k_sharing, sigma = sigma )
data.update({'learning_rate' :  learning_rate, 'epochs' :  epochs,\
             'subordinate' :  subordinate,"patience" : patience,\
             'drop_SGD' : drop_SGD,"orthodox_dt" : orthodox_dt,'with_phase' : with_phase,
             "batch_size" : batch_size, "save_parameter_hist" : save_parameter_hist })
```

We are finally ready to train the model. We do so using the class Binary_Phase_Separation

```
Model = Binary_Phase_Separation()
```

Of which you can learn more about by typing

```
print(Model.__doc__)
```

>>>

```
   This is the main class  of the Phase Separation Binary Classifier (PSBC).
   With its methods one can, aong other things, train the model and
   predict classifications (once the model has been trained).
```

If the above is not enough, you can type

```
print(help(Model))
```

>>>

```
Help on Binary_Phase_Separation in module binary_phase_separation object:

class Binary_Phase_Separation(builtins.object)
 |  Binary_Phase_Separation(cost=None, par_U_model=None, par_P_model=None, par_U_wrt_epochs=None, par_P_w
 |
...
```

But this is maybe too much. So, let's say that you just want to know about how to train. You can get information only about that method

```
print(Model.train.__doc__)
```

>>>

```
        'train' method.

        This method trains the PSBC model with a given set of parameters and
        data.

        Parameters
        ----------
        X : numpy.ndarray of size Nx X N_data
            Matrix with features.
        ...
```

The method that we want is train. So, we do

```
Model.train(
    X_train, Y_train, X_train, Y_train, learning_rate, dt, dx, Nt,\
        weights_k_sharing, eps = eps, epochs = epochs, \
        subordinate = subordinate, with_phase = with_phase,\
        drop_SGD = drop_SGD, sigma = sigma,\
        orthodox_dt = orthodox_dt, print_every = 300,\
            save_parameter_hist = save_parameter_hist
)
```

```
>>>

 epoch : 0 cost 0.11494985702898435

 accuracy : 0.70375

 epoch : 300 cost 0.022553932287346947

 accuracy : 0.9775
```

If you want to take a look at how the cost function behaves over epochs, you can plot it (see full code in Notebook_examples.ipynb). The output is given below.

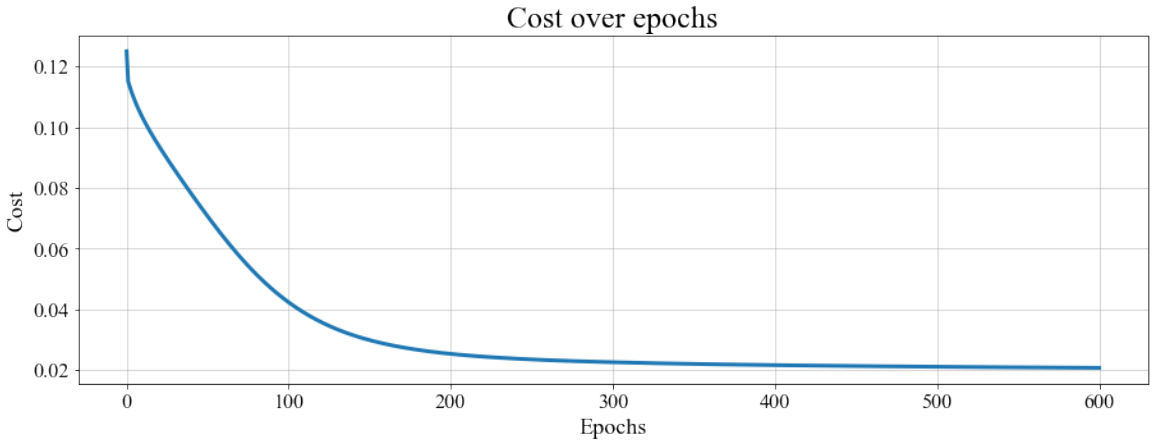


Figure 3.1: The evolution of the cost over epochs, for the 1D PSBC model with labeling 3.2.

And if you want to take a look at the behavior of the set $\mathcal{P}_\alpha$ you can also do. Just type

```
diameter_history = Model.diameters_hist
```

which will give you a dictionary with two keys: “U” and “P”

```
diameter_history.keys()
```

>>>

```
dict_keys(['P', 'U'])
```

They concern the behavior of trainable weights for the U variable, and for the P variable. They can be plotted as

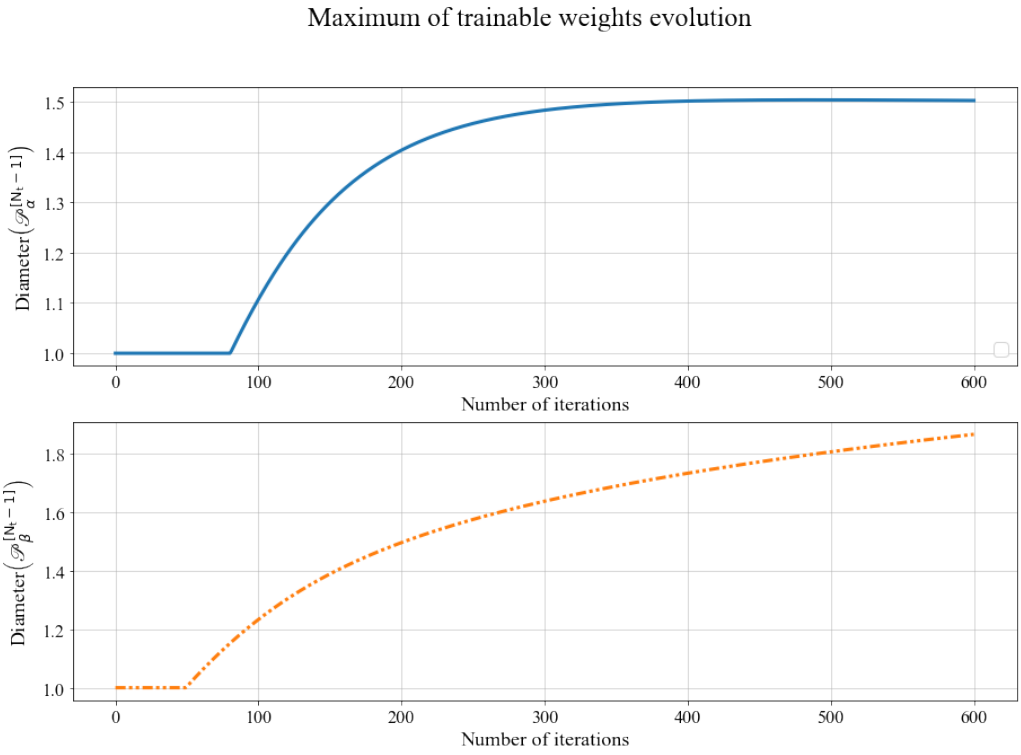


Figure 3.2: The evolution of the diameters $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ over epochs, for the 1D PSBC model with labeling 3.2.

This is the typical behavior of these quantities: they remain constant (equal to 1) up to a certain point, to then grow in a logarithmic shape. Note that the point of departure from the value 1 is different for both variables; that's because both quantities Δ_t^u and Δ_t^p in (3.1) are allowed to vary independently (see the paper for further information).

We can also check the behavior of accuracythroughout epochs:

Note the the model peaks (reaches a point of high accuracy) before the final epoch. This natural "deterioration" is what lead researchers to design **Early Stopping** methods; cf. (Prechelt, 1998). We can in fact know what that epoch was by typing

```
Model.best_epoch
```

>>>

```
array(181)
```

which is before final epoch - in this case, 600. The accucary (for the training set) at epoch 181 was

```
Model.best_accuracy
```

>>>

```
array(1.)
```

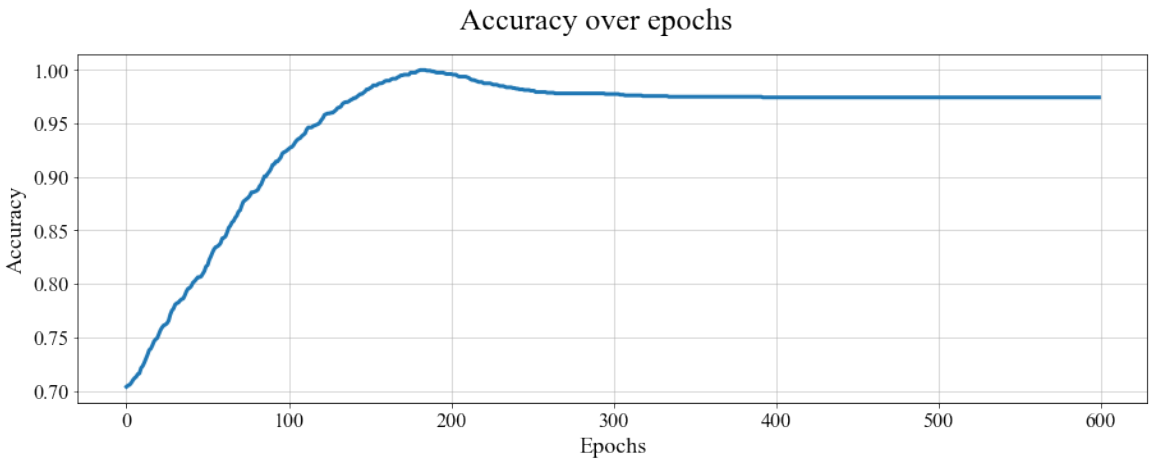


Figure 3.3: The evolution of accuracy over epochs, for the 1D PSBC modelwith labeling 3.2.

that is, 100% accuracy. If you want to retrieve the model parameters at such an epoch you just need to type

```
best_P , best_U = Model.best_par_P_model, Model.best_par_U_model
```

which will give the value of the parameters used when the model achieved its best performance.

For this simple model we did something else: we are saving all the parameters in the model at each epoch. [1]

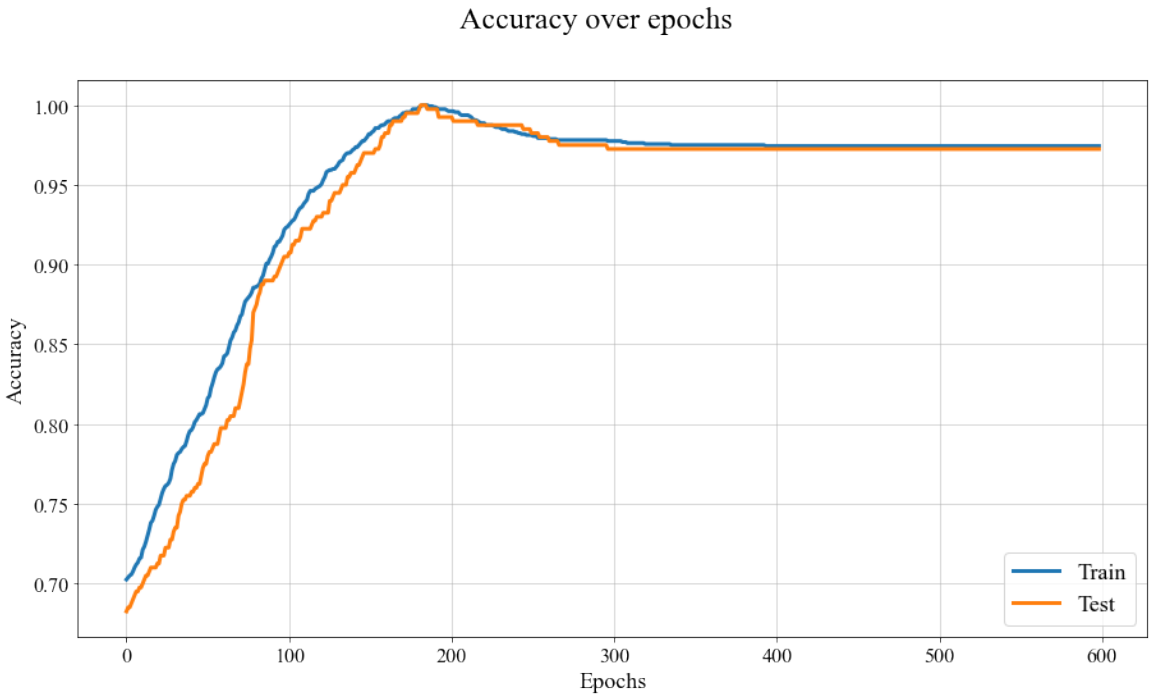


Figure 3.4: Evolution of accuracy for train and test set over epochs, for the 1D PSBC problem with labeling 3.2

[1] It is clear that whenever one deals with big models memory is an impeditive obstruction to reproducing this; in such a case, it is better to set "save_parameter_hist = False" in order to save memory.

Chapter 4

The MNIST database

The MNIST database is a well known database of handwritten digits used in the classical paper (Lecun et al., 1998). In this project we only used (suggestively) the subset of digits "0" and "1", for we are doing binary classification only.

```
### READ MNIST DATASET TO PANDAS DATAFRAME AND THEN TO NUMPY FILE
data_train_MNIST = pd.read_csv('Examples/data_train_normalized_MNIST.csv')
data_test_MNIST = pd.read_csv('Examples/data_test_normalized_MNIST.csv')
```

```
X_train_MNIST = (data_train_MNIST.iloc[:,:-1]).to_numpy()
Y_train_MNIST = np.reshape(data_train_MNIST.iloc[:,-1].to_numpy(), (1,-1))
X_test_MNIST = (data_test_MNIST.iloc[:,:-1]).to_numpy()
Y_test_MNIST = np.reshape(data_test_MNIST.iloc[:,-1].to_numpy(), (1,-1))

X_train_MNIST, X_test_MNIST = X_train_MNIST.T , X_test_MNIST.T
```

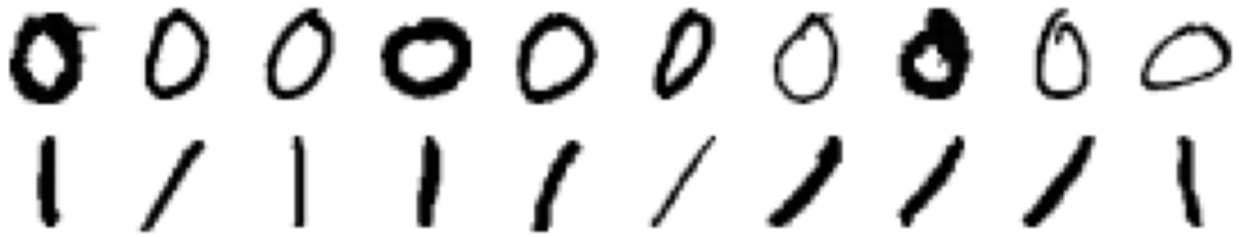


Figure 4.1: A subsample of digits "0" and "1" in the MNIST database.

One can also visualize the trainable weights as heatmaps (see full code in Notebook_examples.ipynb).

4.1 Retrieving some statistics

To get a flavor of what is in the statistics folder, we first need to retrieve some of this data:

```
parameters_MNIST_nondif, stats_folder_MNIST = {}, "Statistics/MNIST/"
with open(stats_folder_MNIST + "parameters_MNIST_nondif.p", 'rb') as fp:
    parameters_MNIST_nondif = pickle.load(fp)

parameters_MNIST_Neumann, stats_folder_MNIST = {}, "Statistics/MNIST/"
with open(stats_folder_MNIST + "parameters_MNIST_Neumann.p", 'rb') as fp:
```

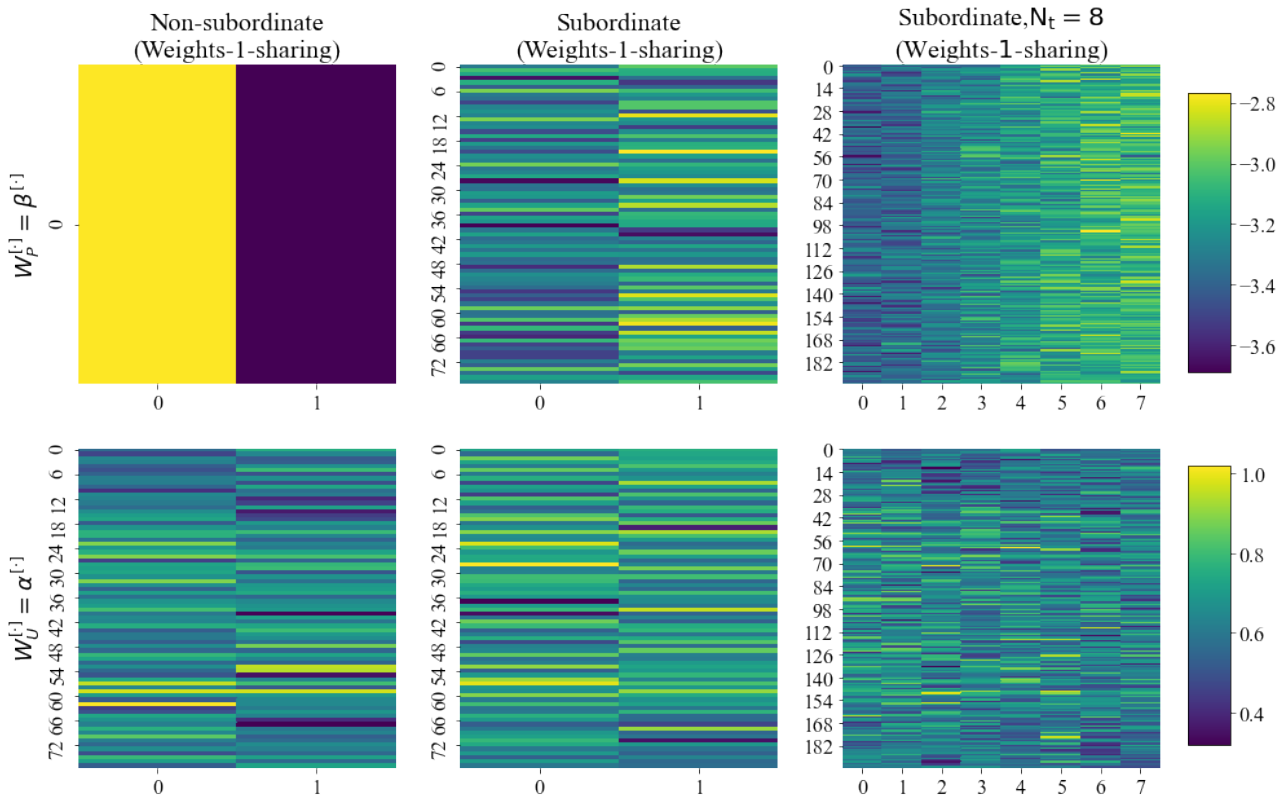


Figure 4.2: Heatmaps of trainable weights for some trained PSBC models in the Examples folder.

```
        parameters_MNIST_Neumann = pickle.load(fp)

parameters_MNIST_Periodic, stats_folder_MNIST = {}, "Statistics/MNIST/"
with open(stats_folder_MNIST + "parameters_MNIST_Periodic.p", 'rb') as fp:
        parameters_MNIST_Periodic = pickle.load(fp)
```

The function *accuracies* is part of the module *aux_fnts_for_jupyter_notebooks.py*, which is available in this Github (Monteiro, 2020b). As before, help on this equation can be called typing 'help(accuracies)'.

With data in the Statistics folder we can plot the graph of accuracies of the non-diffusive PSBC for different values of partition cardinality N_{ptt}.

If can also see the evolution of the maximum of trainable weights over epochs, for a Periodic PSBC with $N_t = 1$.

With these data we can also plot confusion matrices.

```
parent_folder = "Examples/"
folder_now = parent_folder + "W1S-Nt8/simulation1/"
with open(folder_now + "Full_model_properties.p", 'rb') as fp:
        Full_model_properties = pickle.load(fp)
```

There are other things that we do as well: for instance, we can plot Table 4 in the Supplement, which shows the average value of the maximum (in ℓ^∞-norm) of trainable weights of non-diffusive PSBC models for different values of partition cardinality N_t.

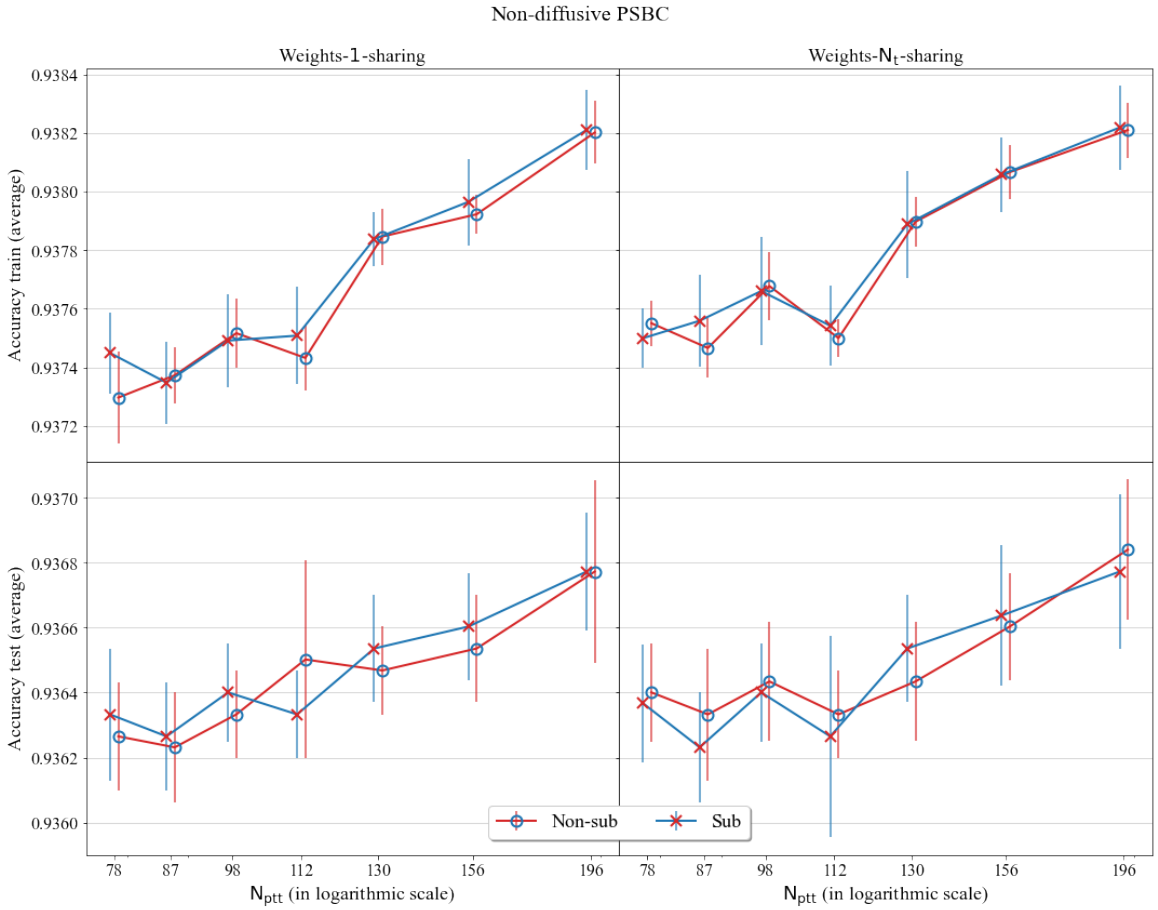


Figure 4.3: A comparison of average accuracy of the non-diffusive PSBC for different values of Partition cardinality; models compared either have subordinate phase (tagged as "Sub") or not (non-subordinate phase, tagged as "Non-sub").

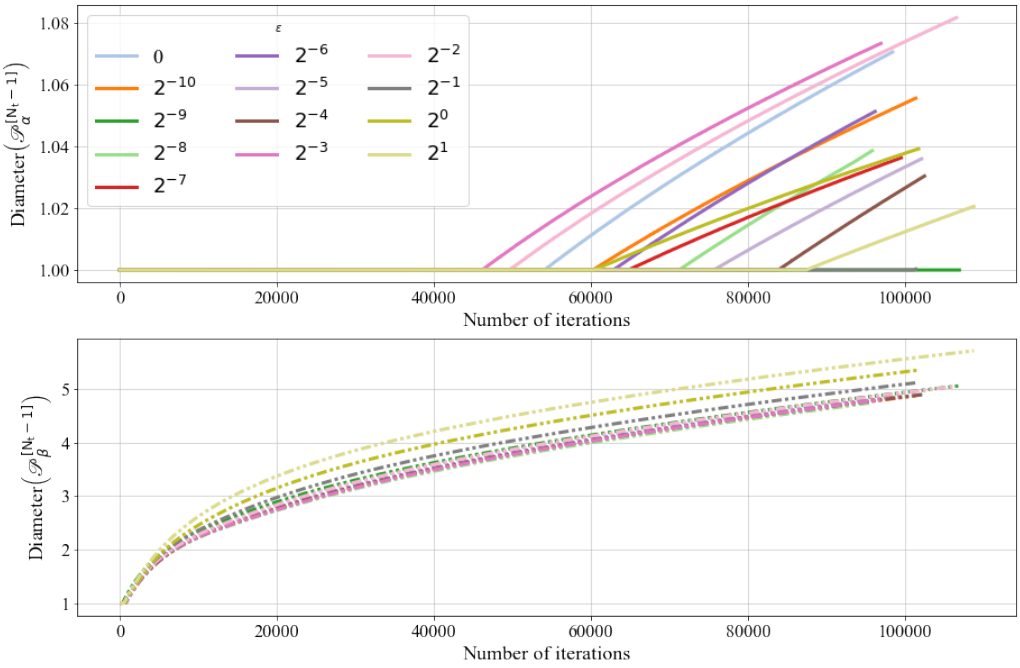


Figure 4.4: The behavior of $\mathcal{P}_\alpha^{[\cdot]}$ and $\mathcal{P}_\beta^{[\cdot]}$ over time, for a Periodic PSBC with $N_t = 1$.

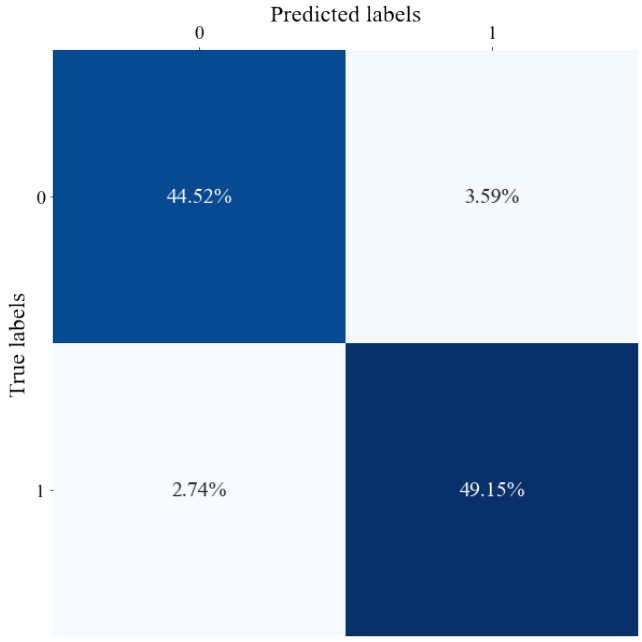


Figure 4.5: Confusion matrix of a realization of the diffusive PSBC with Neumann BCs, weights-1-sharing, and $N_t = 8$.

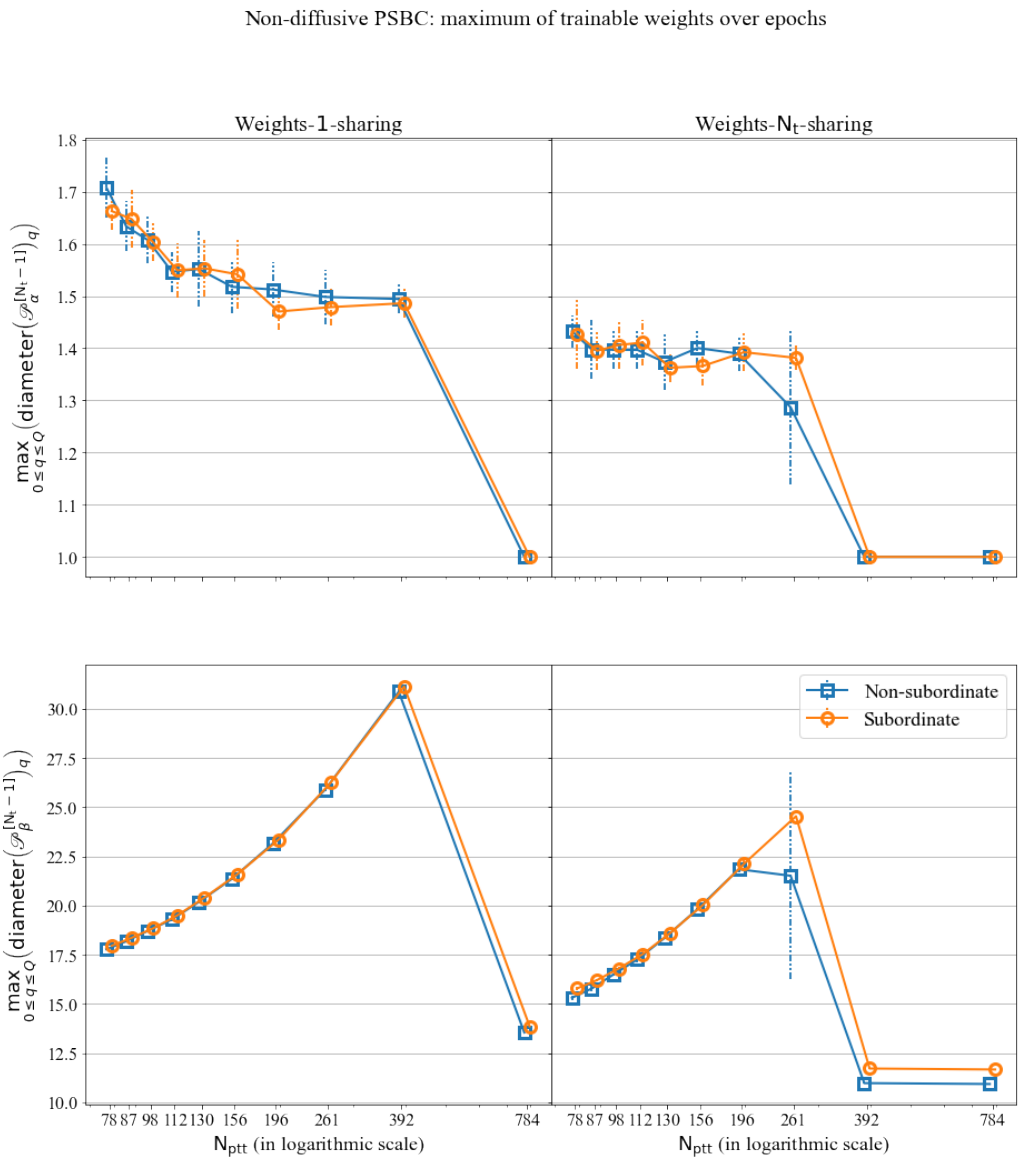


Figure 4.6: The maximum over epochs. You find the code for this plot in Notebook_examples.ipynb

4.2 A "homemade" example: hadwritten 0 and 1

We now step foward to a higher dimensional feature space, using equations that are similar to those shown in Section 3 of this tutorial. There is an interesting interplay between high-dimensionality of feature spaces and model compressibility, which we highlight here by applying the PSBC to the subset "0"-"1" of the MNIST database; we once more refer the reader to Sections 4 and 5 in the paper for more details.

The goal in this section is to illustrate a bit more of the PSBC's use by predicting the label for some of the author's own handwritten numbers. For that we shall use the trainable models available in (Monteiro, 2020a), in the tarball **PSBC_Examples.tar.gz**. Two of these digits are shown below.

Figure 4.7: Two of the author's own handwritten numbers; original photo, treated using Gimp.

In fact, we shall use 6 of the author's handwritten digits - 3 zeros, 3 ones - for this notebook. If you read the first papers of LeCun et al. about the MNIST project, there is a description of the way pictures were taken (see for instance, Section III A in (Lecun et al., 1998), or the explanation in LeCun's MNIST webpage), so that they look the way they do in cell 36 of Notebook_examples.ipynb: the images had to be controlled for angle, centralization, etc; this is part of the statistical design, which I tried to follow here without too much concern (because this is just a tutorial) but, somehow, "as close as possible".

Pictures we cropped using GIMP, a free software for image manipulation: you take a picture, crop it, go to image, set it into grayscale, adjust for light contrast and other things. And that's it.

Now, with them cropped, "MNIST-like" grayscale images in hands, you proceed as in the next cell, reshaping these pictures as a 28 x 28 matrix. We show what the two digits that you saw before will look like.

```
from PIL import Image

def create_MNIST_type_figure(name):
    """Convert jpg figure to a (28,28) numpy array"""
    image = Image.open(name).convert('L')
    image2 = image.resize((28,28))
    im2_as_array = 255- np.array(image2,  dtype=np.uint8)
    print("image has shape", im2_as_array.shape)

    return im2_as_array

my_0 = create_MNIST_type_figure("figures/my_0.jpg")
my_0_v2 = create_MNIST_type_figure("figures/my_0_v2.jpg")
my_0_v3 = create_MNIST_type_figure("figures/my_0_v3.jpg")
my_1 = create_MNIST_type_figure("figures/my_1.jpg")
my_1_v2 = create_MNIST_type_figure("figures/my_1_v2.jpg")
```

```
my_1_v3 = create_MNIST_type_figure("figures/my_1_v3.jpg")

fig, ax = plt.subplots(1,2)
ax[0].imshow(my_0, cmap='binary')
ax[1].imshow(my_1, cmap='binary')
ax[0].axis(False)
ax[1].axis(False)
plt.show()
```

You will get this

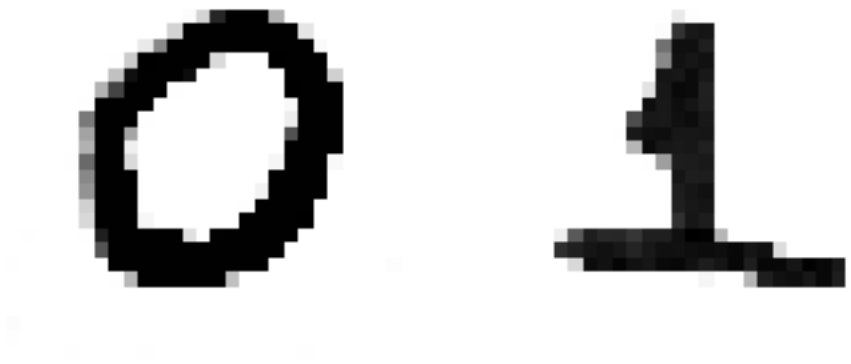


Figure 4.8: The handwritten digits shown above, now as 28X28 pixels images.

Recall that we need to flatten these matrices,

```
my_0_for_psbc = my_0.flatten(order='C')
...
```

and we can then combine all these 6 flattened images as columns in a single matrix **combined_handwritten**, whose size is 784×6.

Now we initialize the PSBC model

```
with open("Examples/W1S-Nt8/simulation1/Full_model_properties.p", 'rb') as fp:
        load_mnist = pickle.load(fp)

psbc_testing = Binary_Phase_Separation()
```

and apply to our matrix/data **combined_handwritten** with the trained weights we have chosen.

```
prediction = psbc_testing.predict(combined_handwritten, load_mnist["best_par_U_model"],
        load_mnist["best_par_P_model"])
print(prediction)
```

...and we fail, as we see in the output below. Floating number overflows, "NAN", etc.. sad news..

```
>>>
[0 0 0 0 0 0]

/Users/rafaelmonteiro/Desktop/PSBC/All_cases/binary_phase_separation.py:475:
RuntimeWarning: overflow encountered in multiply
  v = v +  dt * v * (1-v) * (v-alpha_x_t)
/Users/rafaelmonteiro/Desktop/PSBC/All_cases/binary_phase_separation.py:477:
```

```
RuntimeWarning: invalid value encountered in matmul
  v = np.matmul(Minv,v)
/Users/rafaelmonteiro/Desktop/PSBC/All_cases/binary_phase_separation.py:1128:
RuntimeWarning: invalid value encountered in greater
  keepdims = True, axis = 0))) > .5, dtype = np.int32)
```

This seems really bad... but do not despair: recall that data need to satisfy the **normalization conditions**: all the features have to be in the range $[0, 1]$. We are in fact very far from that now: if you look for the minimum and maximum value of these matrices you will get 0 and 255, respectively, as an answer.

With that said, let's normalize the data:

```
init_data = Initialize_Data()
```

So, the data get's normalized, but centered. By default, it gets rescaled in the range [0.4,0.6]. What we do then is: (i) we normalize it, then (ii) we add 0.1 to it.

```
combined_handwritten_for_psbc, _, _ = init_data.normalize(combined_handwritten)
combined_handwritten_for_psbc = 0.1+ combined_handwritten_for_psbc
```

Now we are in better shape: if you look for minimum and maximum of the matrix *combined_handwritten_for_psbc* you will get 0.5 and 0.7000000000000001.

Figure 4.9: Some of the author's handwritten examples for this tutorial.

Now let's see how well the PSBC does in predicting them (note that, for the subset "0" and "1" of the MNIST database we already know that all these model perform quite well, separating these two classes with accuracy about 94%).

```
for name in ["W1S-NS", "W1S-S", "WNtS-NS", "WNtS-S",\
             "W1S-Nt2", "W1S-Nt4", "W1S-Nt8",\
              "WNtS-Nt1","WNtS-Nt2", "WNtS-Nt4", "WNtS-Nt8",\
              "Per_W1S-Nt2", "Per_W1S-Nt4", "Per_W1S-Nt8",\
              "Per_WNtS-Nt1","Per_WNtS-Nt2", "Per_WNtS-Nt4", "Per_WNtS-Nt8"]:

    with open("Examples/"+name+"/simulation1/Full_model_properties.p", 'rb') as fp:
        load_mnist = pickle.load(fp)

    psbc_testing = Binary_Phase_Separation()
    prediction = \
    psbc_testing.predict(
        combined_handwritten_for_psbc,  load_mnist["best_par_U_model"],\
                        load_mnist["best_par_P_model"],\
        subordinate = load_mnist["best_par_U_model"]["subordinate"]
    )
    print("Model", name, " predicts", np.squeeze(prediction), "and correct is, [0 0 0 1 1 1]" )
```

```
>>>
Model W1S-NS  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model W1S-S  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model WNtS-NS  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
```

```
Model WNtS-S  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model W1S-Nt2  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model W1S-Nt4  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model W1S-Nt8  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model WNtS-Nt1  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model WNtS-Nt2  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model WNtS-Nt4  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model WNtS-Nt8  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_W1S-Nt2  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_W1S-Nt4  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_W1S-Nt8  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_WNtS-Nt1  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_WNtS-Nt2  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_WNtS-Nt4  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
Model Per_WNtS-Nt8  predicts [0 0 0 0 1 1] and correct is, [0 0 0 1 1 1]
```

It is getting all correct, except for the 4th picture (which is in fact the way that I usually write, with that huge horizontal “foot”).

Chapter 5

The Phase Separation Binary Classifier: where to read more about it

There are quite a few places that you can read more about the Phase Separation Binary Classifier (PSBC), and also see further examples.

- I have posted a paper with all the mathematics behind the model. It is quite self contained, and you can "ignore" most of the references that I do to other theories (like my comments about maximum principles in parabolic and elliptic PDEs, etc). There is a preprint currently available on arXiv at https://arxiv.org/abs/2009.02467
- There are many other examples of use in the jupyter-notebooks (for 1D toy problem, for MNIST)
- If you want to have access to the trainable examples I used, and to the computational statistics, you can either download the files PSBC_Examples.tar.gz and PSBC_computational_statistics.tar.gz at the companion data repository to this project at Zenodo. There is also this script in this git-hub that you can use to download all you need automatically. You only need wget and tar to use it.
- In this Github you will also find a manual named README.pdf to the data in the repository, and also the script download_PSBC.shand use it to download everything you need. There are instructions on how to run the script in the README.pdf file.

Bibliography

Allen, S. M. and Cahn, J. W. (1979). A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening. *Acta metallurgica*, 27(6):1085–1095.

Angenent, S. B., Mallet-Paret, J., and Peletier, L. A. (1987). Stable transition layers in a semilinear boundary value problem. *J. Differential Equations*, 67(2):212–242.

Aronson, D. G. and Weinberger, H. F. (1978). Multidimensional nonlinear diffusion arising in population genetics. *Adv. in Math.*, 30(1):33–76.

Dennis, M. R., Glendinning, P., Martin, P. A., Santosa, F., and Tanner, J., editors (2015). *The Princeton companion to applied mathematics*. Princeton University Press, Princeton, NJ.

Fife, P. C. (1979). *Mathematical aspects of reacting and diffusing systems*, volume 28 of *Lecture Notes in Biomathematics*. Springer-Verlag, Berlin-New York.

Hoff, D. (1978). Stability and convergence of finite difference methods for systems of nonlinear reaction-diffusion equations. *SIAM J. Numer. Anal.*, 15(6):1161–1177.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

Monteiro, R. (2020a). Data repository for the paper "Binary classification as a phase separation process". https://dx.doi.org/10.5281/zenodo.4005131.

Monteiro, R. (2020b). Source code for the paper "Binary classification as a phase separation process". https://github.com/rafael-a-monteiro-math/Binary_classification_phase_separation.

Nishiura, Y. (2002). *Far-from-equilibrium dynamics*, volume 209 of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, RI. Translated from the 1999 Japanese original by Kunimochi Sakamoto, Iwanami Series in Modern Mathematics.

Prechelt, L. (1998). *Early Stopping - But When?*, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg.

Smoller, J. (1994). *Shock waves and reaction-diffusion equations*, volume 258 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, New York, second edition.