MICAS901 - Introduction to Optimization
Homework 3: Final Project

Abdelaziz BOUNHAR

November 25, 2020

Remark on Model Complexity choice

The beginning was more like experimental, we tried to implement the different methods, where the model complexity selection was included in the KFold Cross-Validation, to find the best one for each methods. After that, we have chosen one that is a tradeoff between all the methods which is **5** .The comparison of the different methods is done using the same model complexity.

In this project, we make the following assumptions :

- X is a (N, d) matrix.
- $W1$ is a (mc, d) matrix.
- $W2$ is a $(1, mc)$ vector.

Where mc is the model complexity, N the number of observations and d the number of features.

1 Part I: deep linear network

We will start with a 2-layer deep linear network, which is a special case case obtained by setting all activation functions to an identity. Thus, the corresponding training problem is given by the following optimization problem:

$$min_{W_2, W_1} \frac{1}{N} \sum_{i=1}^{N} \| W_2 W_1 x_i - y_i \|_2^2 + \lambda_1 \| W_1 \|_F^2 + \lambda_2 \| W_2 \|_F^2$$

Derivation of the gradient

From the optimization problem described above, we have the loss function defined as follow :

$$f(W_1, W_2) = \frac{1}{N} \sum_{i=1}^{N} \| W_2 W_1 x_i - y_i \|_2^2 + \lambda_1 \| W_1 \|_F^2 + \lambda_2 \| W_2 \|_F^2$$

Which can be written in the matrix form in order to simplify the implementation as :

$$f(W_1, W_2) = \frac{1}{N} \left\| W_2 W_1 X^T - Y \right\|_2^2 + \lambda_1 \left\| W_1 \right\|_F^2 + \lambda_2 \left\| W_2 \right\|_F^2$$

First we should derive the gradient with respect to each W_i.

1. **With respect to W_2 :**

   We have that $\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2} [\frac{1}{N} \left\| W_2 W_1 X^T - Y \right\|_2^2 + \lambda_1 \left\| W_1 \right\|_F^2 + \lambda_2 \left\| W_2 \right\|_F^2]$

   Therefore :

   $\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2} [\frac{1}{N} \left\| W_2 W_1 X^T - Y \right\|_2^2] + 2\lambda_2 W_2$

   By putting $Z_2 = W_2 W_1 X^T$ we can write :

   $$\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2} [\frac{1}{N} \left\| Z_2 - Y \right\|_2^2] + 2\lambda_2 W_2$$

   From the Chain Rule of derivative, one can have :

   $$\nabla_{W_2} f(W_1, W_2) = \nabla_{Z_2} [\frac{1}{N} \left\| Z_2 - Y \right\|_2^2] \nabla_{W_2} Z_2 + 2\lambda_2 W_2$$

   Hence :

   $$\boxed{\nabla_{W_2} f(W_1, W_2) = \frac{2}{N} (W_2 W_1 X^T - Y)(W_1 X^T)^T + 2\lambda_2 W_2}$$

2. **With respect to W_1 :**

   We have that $\nabla_{W_1} f(W_1, W_2) = \nabla_{W_1} [\frac{1}{N} \left\| W_2 W_1 X^T - Y \right\|_2^2 + \lambda_1 \left\| W_1 \right\|_F^2 + \lambda_2 \left\| W_2 \right\|_F^2]$

   By putting $Z_2 = W_2 W_1 X^T = W_2 Z_1$ where $Z_1 = W_1 X^T$ we can write :

   $$\nabla_{W_1} f(W_1, W_2) = \nabla_{W_1} [\frac{1}{N} \left\| Z_2 - Y \right\|_2^2] + 2\lambda_1 W_1$$

   From the Chain Rule of derivative, one can have :

   $$\nabla_{W_1} f(W_1, W_2) = \nabla_{Z_2} [\frac{1}{N} \left\| Z_2 - Y \right\|_2^2] \nabla_{Z_1} Z_2 \nabla_{W_1} Z_1 + 2\lambda_1 W_1$$

   $$\nabla_{W_1} f(W_1, W_2) = \nabla_{Z_2} [\frac{1}{N} \left\| Z_2 - Y \right\|_2^2] \nabla_{Z_1} W_2 Z_1 \nabla_{W_1} W_1 X^T + 2\lambda_1 W_1$$

   Hence :

   $$\boxed{\nabla_{W_1} f(W_1, W_2) = \frac{2}{N} W_2^T [(W_2 W_1 X^T - Y) X] + 2\lambda_1 W_1}$$

The implementation of this is defined as follow :

```
def computePartialDerivative(X, Y, W1, W2, lambda1, lambda2) :

    W1_W2 = np.dot(W2, W1)
    ffwd = (2/X.shape[0])*(np.dot(W1_W2, X).T - Y)

    # for grad_1
    tmp_1 = np.dot(W2.T, ffwd.T)
    tmp_2 = np.dot(tmp_1, X.T)

    # for grad 2
    W1_XT = np.dot(W1, X)

    # compute the final result
    grad_1 = tmp_2 + 2*lambda1*W1
    grad_2 = np.dot(ffwd.T, W1_XT.T) + 2*lambda2*W2

    return grad_1, grad_2


def computeLoss(X, Y, W1, W2, lambda1, lambda2) :

    W1_W2 = np.dot(W2, W1)
    ffwd = np.dot(W1_W2, X) - Y

    norm_1 =  (1/X.shape[0])*np.linalg.norm(ffwd)
    norm_2 = lambda1*np.linalg.norm(W1)
    norm_3 = lambda2*np.linalg.norm(W2)

    return norm_1 + norm_2 + norm_3

def MSE(loss):
    return np.sum((loss)**2)/(len(loss))
```

a) Back-Propagation with constant step-size

The back-propagation algorithm consist on updating the weights using this generalized formulae :

$$W_i^{k+1} = W_i^k - \alpha_k \nabla_{W_i} f(W_i^k)$$

In our case, the step-size α_k is constant, and f depends on two variables W_1, W_2, therefore the update equations are as follows :

$$\begin{cases} W_1^{(k+1)} = W_1^{(k)} - \alpha \nabla_{W_1} f(W_1^{(k)}, W_2^{(k)}) \\ W_2^{(k+1)} = W_2^{(k)} - \alpha \nabla_{W_2} f(W_1^{(k)}, W_2^{(k)}) \end{cases}$$

Where :

- The gradients $\nabla_{W_1} f(W_1^{(k)}, W_2^{(k)})$ and $\nabla_{W_2} f(W_1^{(k)}, W_2^{(k)})$ are the one we derived before.
- α the step-size we will define using KFold cross-validation.
- The parameters λ_1, λ_2 present in the gradient will also be defined using KFold cross-validation.

This is achieved by the following algorithm :

```python
def backPropagation(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, alpha, N) :

    lossArray = np.zeros(N)

    for i in tqdm(range(N)) :

        loss = computeLoss(X_test, y_test, W1, W2, lambda1, lambda2)

        lossArray[i] = loss

        grad_1, grad_2 = computePartialDerivative(X_train, y_train, W1, W2, lambda1, lambda2)

        W1 = W1 - alpha*grad_1
        W2 = W2 - alpha*grad_2

    print("W2.shape : ", W2.shape)
    print("W1.shape : ", W1.shape)
    print("X.shape : ", X.shape)
    print("Loss mean : ", lossArray.mean())
    return W1, W2, lossArray
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

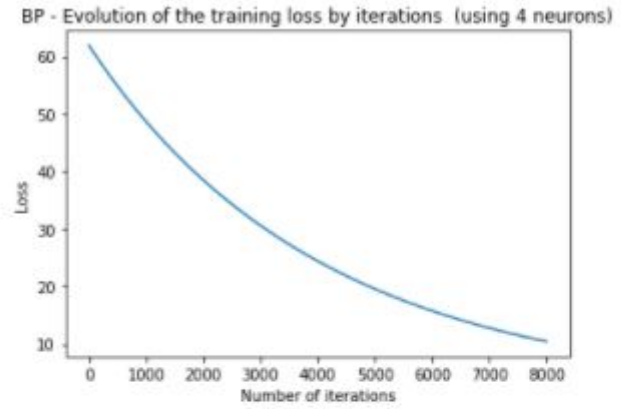
- $\lambda_1 = 0.01$
- $\lambda_2 = 0.08$
- $\alpha = 2 \cdot 10^{-7}$
- Number of Neurons = 3

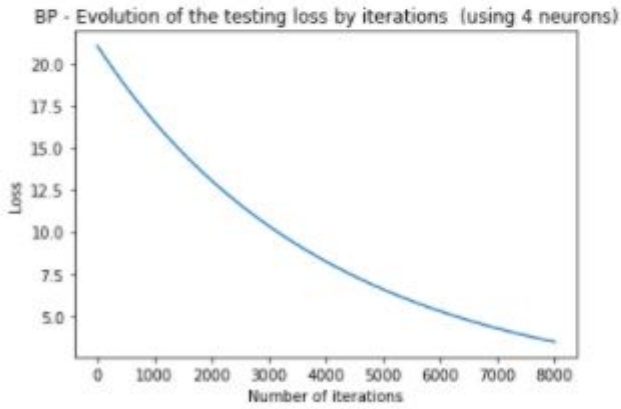
```python
l1_, l2_, alpha, n_neuron = getBestParams(lossCV_BP, best_valCV_BP)
```

```
len(lossTestingKFold) =  3200
len(best_val) =  3200
Best lambda1 :  0.01
Best lambda2 :  0.08
Best alpha :  2e-07
Best n_neuron :  3
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 10.43
- Testing loss : 3.49
- Training MSE : 998.83
- Testing MSE : 114.67






```
The training MSE is : 998.8342958326778
The testing MSE is : 114.6703543837977
```

b) Mini-Batch SGD

The mini-batch SGD improves the gradient estimate compared to plain SGD. Actually the parameters are updated after computing the gradient of error with respect to a subset of the training set. Since we consider only a subset of training set, the updates of the model parameters are made faster especially when using the matrix/vectorized form of the optimization problem. Furthermore, mini-batch SGD is a solution to noisy gradient estimate in SGD, it makes the updates less noisy depending on the batch-size.

Similarly to back-propagation, the formulae for updating the weights are defined as follow :

$$\begin{cases} W_1^{(k+1)} = W_1^{(k)} - \alpha \nabla_{W_1} f(W_1^{(k)}, W_2^{(k)}) \\ W_2^{(k+1)} = W_2^{(k)} - \alpha \nabla_{W_2} f(W_1^{(k)}, W_2^{(k)}) \end{cases}$$

Where :

- The gradients $\nabla_{W_1} f(W_1^{(k)}, W_2^{(k)})$ and $\nabla_{W_2} f(W_1^{(k)}, W_2^{(k)})$ are the one we derived before.
- α the step-size we will define using KFold cross-validation.
- The parameters λ_1, λ_2 present in the gradient will also be defined using KFold cross-validation.

However in this case, rather than computing the gradient over the whole training set, we only pick a mini-batch corresponding to a predefined batch-size.

This is achieved by the following algorithm :

```
def miniBatchSGD(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, alpha, N, batch_size, with_print) :

    lossArray = np.zeros(N)

    # just to avoid prints when we do cross validation
    if(with_print) :

        for i in tqdm(range(N)) :

            mini_batches_train = create_mini_batches(X_train, y_train, batch_size)

            for mini_batch in mini_batches_train:

                X_train, y_train = mini_batch
                loss = computeLoss(X_test.T, y_test, W1, W2, lambda1, lambda2)

                lossArray[i] = loss

                grad_1, grad_2 = computePartialDerivative(X_train.T, y_train, W1, W2, lambda1, lambda2)

                W1 = W1 - alpha*grad_1
                W2 = W2 - alpha*grad_2

        print("W2.shape : ", W2.shape)
        print("W1.shape : ", W1.shape)
        print("X.shape : ", X.shape)
        print("Loss mean : ", lossArray.mean())
        print("Last loss value : ", lossArray[-1])
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

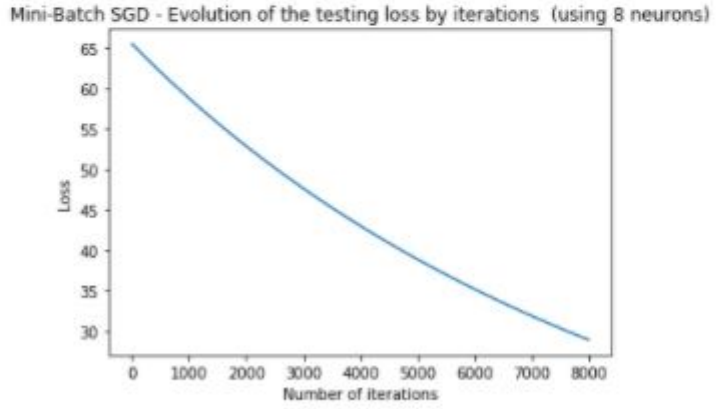
- $\lambda_1 = 0.002$
- $\lambda_2 = 0.004$
- $\alpha = 2 \cdot 10^{-7}$
- Number of Neurons = 8

```
l1_SGD, l2_SGD, alpha_SGD, n_neuron_ = getBestParams(lossCV_SGD_1, best_valCV_SGD_1)
```

```
len(lossTestingKFold) =  3200
len(best_val) =  3200
Best lambda1 :  0.002
Best lambda2 :  0.004
Best alpha :  2e-07
Best n_neuron :  8
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 0.4
- Testing loss : 1.19
- Training MSE : 199.32
- Testing MSE : 2078.16




```
The training MSE is : 199.3264496138547
The testing MSE is : 2078.164398521288
```

Method to select the batch size

Full batch learners must perform the full training set scan for every single weight update. Mini-batch learners get to perform that same weight update multiple times per training set scan.

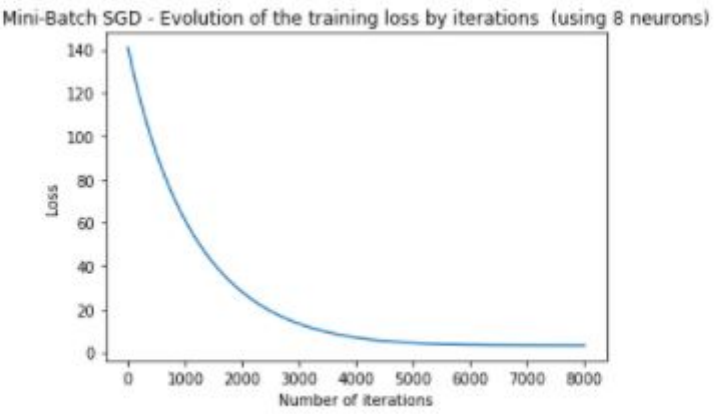
When the batch-size increases, each mini-batch gradient may contain more redundant information, we shall then find a **tradeoff**.

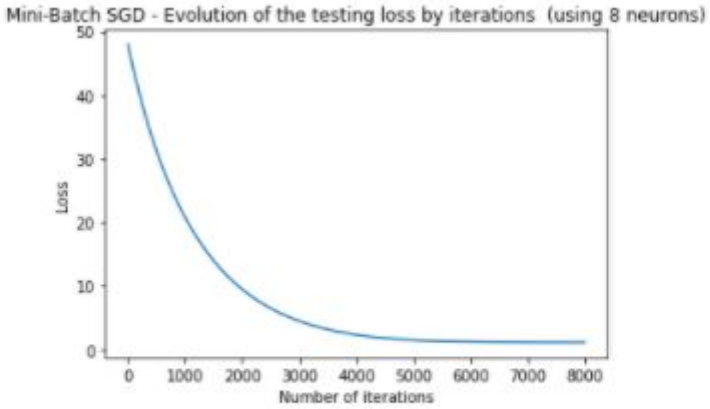
When using large batch-size, we are more like in the back-propagation case (full batch learner). Moreover, full batch learners will always make smooth steps towards the globally optimal decision point because it's aligned to the gradient, thus there's no randomness compared to the case where we have smaller batch-size. This later is more exposed to randomness in the training set and therefore resulting in weight steps that look significantly more random than full batch steps. The smaller the batch size, the greater the randomness.

On the other hand, adding N more examples into a mini-batch, results in reducing the uncertaincy in the gradient by a factor of only $O(\sqrt{N})$. Hence, the batch-size should be a factor of 2 (2^n), where n is chosen according to the training set size. It is more common to choose $n = 5$ or $n = 6$ resulting in a batch-size of 32 or 64. In fact those two values for the batch-size has proved to be efficient from empirical tests.

Sanity check

Theoretically, setting the batch size equal to the training set size, we must have the same behavior as in the Back-propagation algorithm since we compute the gradient over a mini-batch that has batch-size equal to the size of the training set.






```
The training MSE is : 1496.8803827572622
The testing MSE is : 173.15399152457567
```

We can remark that with a full batch SGD, we achieve the same performance as a Back-Propagation. We can also remark that we got a higher testing MSE, this can be seen as the SGD didn't find the optimal solution due to the batched gradient computation, which can be normal in this case.

c) ADAM

The ADAM optimization algorithm is an extension to SGD. ADAM tries to find a better estimate for the true gradient by adding First and Second order Moment, then models the bias and moves along the bias of the First order Moment rather than with the gradient.

Since we have two weights, the steps are as follow :

It starts by computing the two gradients $g1^{(k)} = \nabla_{W_1} f$ and $g2^{(k)} = \nabla_{W_2} f$ of a random sample X_i. Those gradients are the one we derived in the beginning of this part.

It approximate bias for First order Moment is then found by :

$$U1^{(k+1)} = \beta_1 U1^{(k)} + (1 - \beta_1) g1^{(k)}$$

$$U2^{(k+1)} = \beta_1 U2^{(k)} + (1 - \beta_1) g2^{(k)}$$

It approximate bias for Second order Moment is then found by :

$$V1^{(k+1)} = \beta_2 V1^{(k)} + (1 - \beta_2) g1^{(k)} \circ g2^{(k)}$$

$$V2^{(k+1)} = \beta_2 V2^{(k)} + (1 - \beta_2) g2^{(k)} \circ g2^{(k)}$$

Then computes bias term of First order Moment :

$$BiasU1^{(k+1)} = U1^{(k)}/(1 - \beta_1^k)$$
$$BiasU2^{(k+1)} = U2^{(k)}/(1 - \beta_1^k)$$

And the bias term of Second order Moment :

$$BiasV1^{(k+1)} = V1^{(k)}/(1 - \beta_2^k)$$
$$BiasV2^{(k+1)} = V2^{(k)}/(1 - \beta_2^k)$$

After all, the weights gets updated by the following formulae :

$$W1^{(k+1)} = W1^{(k)} - ((BiasV1^{(k+1)})^{\frac{1}{2}} + \delta.1)^{-1} \circ BiasU1^{(k+1)}$$

$$W2^{(k+1)} = W2^{(k)} - ((BiasV2^{(k+1)})^{\frac{1}{2}} + \delta.1)^{-1} \circ BiasU2^{(k+1)}$$

This can be achieved by the following algorithm :

```
def ADAM(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, beta1, beta2, n_neuron, N, d, use_one_rs, with_print)

    lossArray = np.zeros(N)

    # For W1
    U1 = np.zeros((n_neuron,d))
    V1 = np.zeros((n_neuron,d))

    # For W2
    U2 = np.zeros((1,n_neuron))
    V2 = np.zeros((1,n_neuron))

    # delta to prevent any division by zero
    delta = 1e-5


    if (with_print):

        for i in tqdm(range(N)) :

            # Compute Loss on prediction
            loss = computeLoss(X_test.T, y_test, W1, W2, lambda1, lambda2)
            lossArray[i] = loss

            if (use_one_rs):

                # compute gradient for one random sample i
                random_sample_indice = np.random.randint(X_train.shape[0])
                random_X = X_train[random_sample_indice].reshape(-1,1)
                random_y = y_train[random_sample_indice].reshape(-1,1)

                grad_1, grad_2 = computePartialDerivative(random_X, random_y, W1, W2, lambda1, lambda2)

            else :
                # compute gradient on full X_train
                grad_1, grad_2 = computePartialDerivative(X_train.T, y_train, W1, W2, lambda1, lambda2)

            # approximate bias for FoM
            U1 = beta1*U1 + (1-beta1)*grad_1
            U2 = beta1*U2 + (1-beta1)*grad_2

            # approximate bias for SoM
            V1 = beta2*V1 + (1-beta2)*grad_1*grad_1
            V2 = beta2*V2 + (1-beta2)*grad_2*grad_2

           # compute bias term of FoM
           x1 = 1- (beta1**i) + delta
           bias_U1 = U1/x1
           bias_U2 = U2/x1

           #compute bias term of SoM
           x2 = 1- (beta2**i) + delta
           bias_V1 = V1/x2
           bias_V2 = V2/x2

           # update weights
           inverse_1 = np.linalg.pinv((np.sqrt(bias_V1) + delta))
           inverse_2 = np.linalg.pinv((np.sqrt(bias_V2) + delta))

           W1 = W1 - inverse_1.T*bias_U1
           W2 = W2 - (inverse_2*bias_U2.T).T

           # to avoid the blowing weight effect
           minmax = MinMaxScaler()
           W1 = minmax.fit_transform(W1)
           minmax = MinMaxScaler()
           W2 = minmax.fit_transform(W2)

       print("W2.shape : ", W2.shape)
       print("W1.shape : ", W1.shape)
       print("X.shape : ", X.shape)
       print("Loss mean : ", lossArray.mean())
   return W1, W2, lossArray
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

- $\lambda_1 = 0.002$
- $\lambda_2 = 0.004$

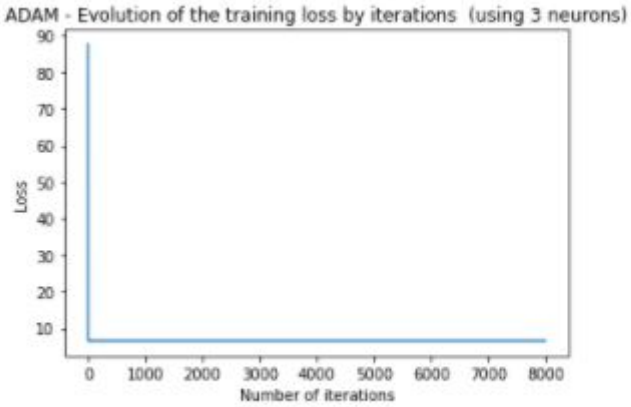
- $\beta_1 = 0.1$
- $\beta_2 = 0.1$
- Number of Neurons = 3

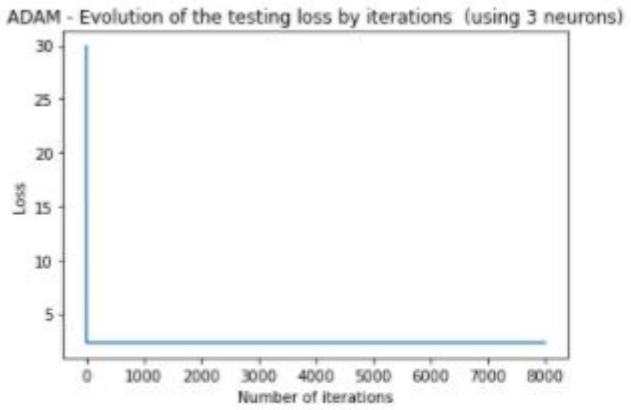
```
l1_ADAM_1, l2_ADAM_1, beta1_ADAM_1, beta2_ADAM_1, n_neuron_ = getBestParams(lossCV_ADAM_1, best_valCV_ADAM_1)
```

```
len(lossTestingKFold) =  5760
len(best_val) =  5760
Best lambda1 :  0.002
Best lambda2 :  0.004
Best beta 1 :  0.1
Best beta 2 :  0.1
Best n_neuron :  3
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 6.54
- Testing loss : 2.28
- Training MSE : 43.82
- Testing MSE : 5.35






```
The training MSE is : 43.82568709721892
The testing MSE is : 5.350807674177033
```

d) BCD

Let's start by deriving each BCD sub-problem, and the resulting update equation for each layer.

We have the following optimization problem :

$$f(W_1, W_2, X, Y) = \frac{1}{N} \| W_2 W_1 X - Y \|_2^2 + \lambda_1 \| W_1 \|_2^2 + \lambda_2 \| W_2 \|_2^2$$

The two BCD sub-problem for this optimization problem are :

$$min_{W1} = \frac{1}{N} \| W_2 W_1 X - Y \|_2^2 + \lambda_1 \| W_1 \|_2^2$$

$$min_{W2} = \frac{1}{N} \| W_2 W_1 X - Y \|_2^2 + \lambda_2 \| W_2 \|_2^2$$

To find a closed form solution, we should start by derivating each sub-problem with respect to W_i.

For W_1 :

$$\nabla_{W1} f = \frac{2}{N} (W_1)^T (W_2 W_1 X - Y)(X)^T + 2\lambda_1 W_1$$

$$\nabla_{W1} f = 0 <=> W_2((W_1 X)(W_1 X)^T + \lambda_2 I_d) = Y(W_1 X)^T$$

$$W_2^T W_2 W_1 X X^T + \lambda_1 W_1 - W_2^T Y X^T = 0$$

$$W_2^T W_2 W_1 X X^T + \lambda_1 W_1 = W_2^T Y X^T$$

However, here **we encounter a problem**; $W_2^T W_2 W_1 X X^T$ is a (N, N) matrix, and $\lambda_1 W_1$ is a (N, d) matrix, and mathematically, this addition is **impossible**.

For W_2 :

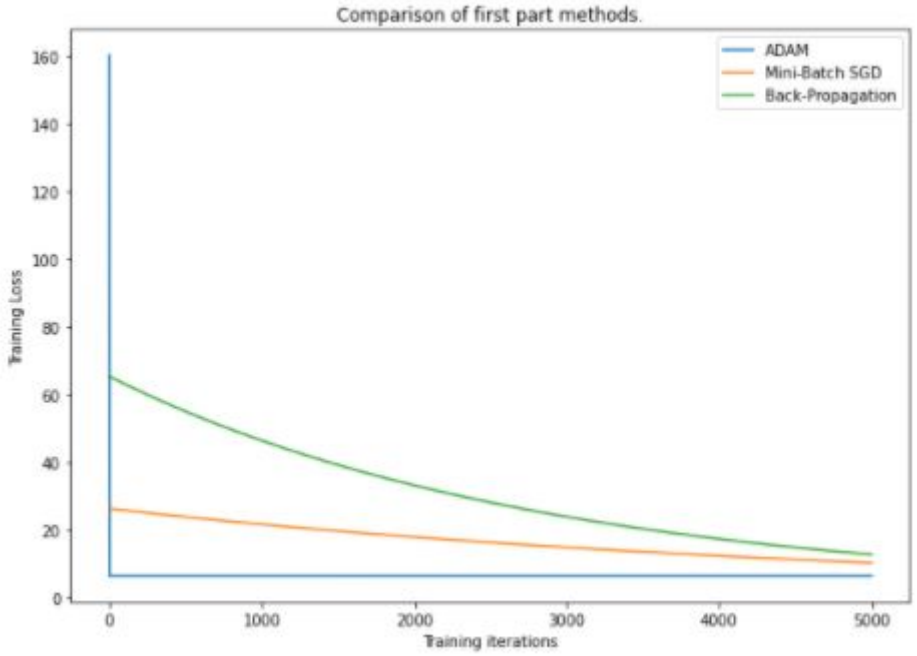
$$\nabla_{W2} f = \frac{2}{N} (W_2 W_1 X - Y)(W_1 X)^T + 2\lambda_2 W_2$$

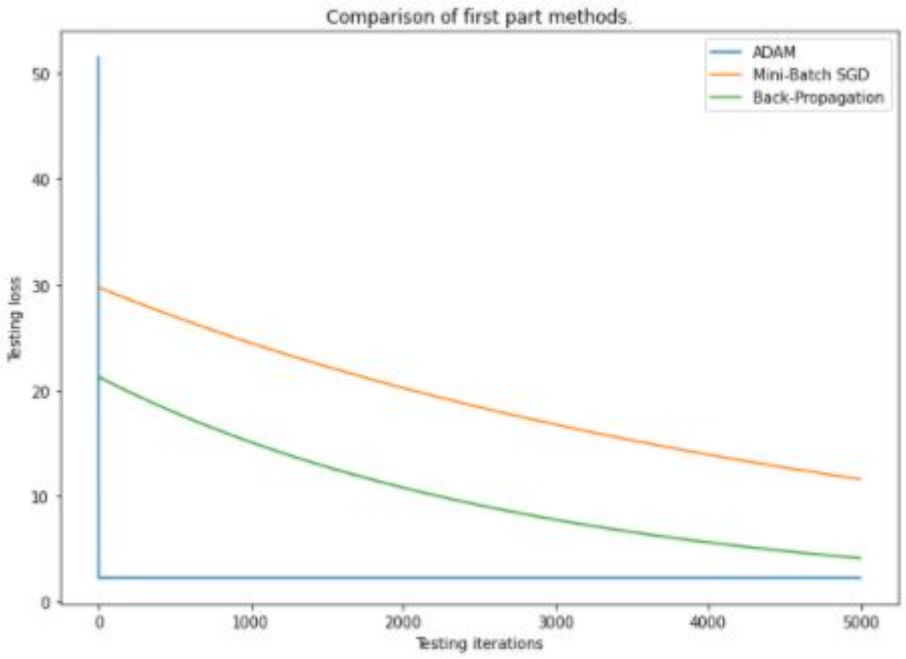
$$\nabla_{W2} f = 0 <=> W_2((W_1 X)(W_1 X)^T + \lambda_2 I_d) = Y(W_1 X)^T$$

$$\boxed{W_{2opt} = Y(W_1 X)^T((W_1 X)(W_1 X)^T + \lambda_2 I_d)^{-1}}$$

Remark : I couldn't derive a closed form solution for the gradient with respect to W_1 after many tries (matrix decomposition, playing around with some lemmas... but in vain). Therefore, I implemented a version of BCD that use the gradient descent, however in this case, we will be doing a back-propagation. Hence, I just tried it, the method is present in the notebook of the project but it will not be present in this section. Actually, after doing all the project, I got the point that we can derive a closed form solution when the activation function is the identity, but not in another case which is the idea of BCD in part 2.

e) Training and test error after after convergence is reached





Comments on results

We can see that ADAM optimizer is the one that converges faster, almost instantaneous

compared to the other methods. Furthermore, Mini-Batch SGD converges also faster in training due to the fact that it doesn't catch the noise from the gradient. However, the Back-propagation method is better than the Mini-Batch SGD by a factor of $\frac{1}{3}$ during testing.

We can resume all the results in the table bellow :

	Training loss	**Testing loss**	**Training MSE**	**Testing MSE**	**KFold CV for MSE (K=5)**
Back-Propagation	10.43	3.49	1226.88	128.87	4.05
Mini-Batch SGD	10.26	9.62	311.38	393.08	83.78
ADAM	6.54	2.28	48.5	5.61	3.16
BCD	-	-	-	-	-

As the figure of testing loss evolution over the iterations, we concluded that Back-Propagation was better than Mini-Batch SGD and the MSE computed using a **KFold Cross Validation** with $K = 5$ shows that also.

f) Computational complexity - Complexity of hyperparameter tuning

The computational complexity in this context depends majorly on the computation of matrix products and matrix inverses.

We recall that :

- X is a (N, d) matrix.
- $W1$ is a (mc, d) matrix.
- $W2$ is a $(1, mc)$ vector.

Therefore we have :

- Back-Propagation : The predominant term is $(W_2 W_1 X - Y)(W_1 X)^T$ which is a matrix product between a $(1, N)$ matrix and a (N, d) matrix. Hence the computational complexity is $O(N \cdot d)$.
- Mini-Batch SGD : Same again as Back-Propagation but we compute the gradient for a mini-batch, therefore it is a matrix product between a $(1, mb)$ matrix and a (mb, d) matrix. Hence the computational complexity is $O(mb \cdot d)$.
- ADAM : ADAM involves computation of gradient which is the predominant term in complexity, hence it's like in Back-Propagation and has a computational complexity of $O(N \cdot d)$
- BCD : By intuition, the predominant term will be XX^T (in fact it's the one that gives the bigger matrix) which is a matrix product between a (N, d) matrix and a (d, N) matrix, therefore, the computational complexity is $O(N^2)$.

	Computational complexity	Complexity of hyperparameter tuning
Back-Propagation	$O(N \cdot d)$	$O(p^3)$
Mini-Batch SGD	$O(mb \cdot d)$	$O(p^3)$
ADAM	$O(N \cdot d)$	$O(p^4)$
BCD	$O(N^2)$	$O(1)$

Where p is the number of possible value for each parameter.

g) Using Cross Entropy loss in Back-Propagation

The cross entropy loss function c is defined as :

$$c(X, Y, W) := -\frac{1}{N} \sum_i (y_i \log(f_W(x_i)) + (1 - y_i) \log(1 - f_W(x_i)))$$

Applying the chain rule of derivation, one can have :

$$\boxed{\nabla c(X, Y, W) = -\frac{1}{N} \sum_i \frac{y_i}{f_W(x_i)} \nabla f_W(x_i) + \frac{1-y_i}{1-f_W(x_i)} \nabla f_W(x_i)}$$

Which can be written in matrix form as :

$$\boxed{\nabla c(X, Y, W) = -\frac{1}{N} \nabla f_W(X)[Y f_W(X)^{-1} + (1 - Y)(1 - f_W(X))^{-1}]}$$

Hence, the gradient of the cross entropy loss function is a function of the true gradient of f. We can write that $\nabla c(X, Y, W) = \beta \nabla f_W(x_i)$ therefore, using cross entropy loss function will not have a significant impact unless β makes $\nabla c(X, Y, W)$ converges faster.

2 Part II: deep neural network

Now we the following logistic activation, where $\sigma(z) = \frac{1}{1+exp(-z)}$ at each layer.

Therefore it's derivative is $\sigma'(z) = \frac{exp(-z)}{(1+exp(-z))^2}$

We start by doing a normalization of the training set using the Min-Max Scaler, in order to take into account the fact that the logistic activation compresses its output between $[0, 1]$.

Thus, the optimization problem considered here is the following:

$$min_{W_2,W_1} f(W_2, W_1) = min_{W_2,W_1} \frac{1}{N} \sum_{i=1}^{N} \|\sigma(W_2\sigma(W_1 x_i)) - y_i\|_2^2 + \lambda_1 \|W_1\|_F^2 + \lambda_2 \|W_2\|_F^2$$

$f(W_2, W_1)$ can be written in a matrix form in order to simplify the implementation as :

$$f(W_2, W_1) = \frac{1}{N} \left\|\sigma(W_2\sigma(W_1 X^T)) - Y\right\|_2^2 + \lambda_1 \|W_1\|_F^2 + \lambda_2 \|W_2\|_F^2$$

Remark In this part, for all methods, the equations for updating the weights remains all the same. Therefore, we only need to derive the new gradient with respect to each W_i, and re-implement the methods according to the new gradient formulas.

Derivation of the gradient

Same reasoning as in the first part, we should first derive the gradient with respect to each W_i.

1. **With respect to W_2 :**

   We have that $\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2}[\frac{1}{N} \left\|\sigma(W_2\sigma(W_1 X^T)) - Y\right\|_2^2 + \lambda_1 \|W_1\|_F^2 + \lambda_2 \|W_2\|_F^2]$

   Therefore :

   $\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2}[\frac{1}{N} \left\|\sigma(W_2\sigma(W_1 X^T)) - Y\right\|_2^2] + 2\lambda_2 W_2$

   By putting $Z_2 = \sigma(W_2\sigma(W_1 X^T))$ we can write :

   $$\nabla_{W_2} f(W_1, W_2) = \nabla_{W_2}[\frac{1}{N} \|Z_2 - Y\|_2^2] + 2\lambda_2 W_2$$

   From the Chain Rule of derivative, one can have :

   $$\nabla_{W_2} f(W_1, W_2) = \nabla_{Z_2}[\frac{1}{N} \|Z_2 - Y\|_2^2] \nabla_{W_2} Z_2 + 2\lambda_2 W_2$$

   Hence :

   $$\boxed{\nabla_{W_2} f(W_1, W_2) = \frac{2}{N}[(\sigma(W_2\sigma(W_1 X^T)) - Y) \circ \nabla\sigma(W_2\sigma(W_1 X^T))]\sigma(W_1 X^T)^T + 2\lambda_2 W_2}$$

2. **With respect to W_1 :**

We have that $\nabla_{W_1} f(W_1, W_2) = \nabla_{W_1}[\frac{1}{N}\left\|\sigma(W_2\sigma(W_1X^T)) - Y\right\|_2^2 + \lambda_1 \|W_1\|_F^2 + \lambda_2 \|W_2\|_F^2]$

By putting $Z_2 = \sigma(W_2\sigma(W_1X^T)) = \sigma(W_2Z_1)$ where $Z_1 = \sigma(W_1X^T)$ we can write :

$$\nabla_{W_1} f(W_1, W_2) = \nabla_{W_1}[\frac{1}{N}\|Z_2 - Y\|_2^2] + 2\lambda_1 W_1$$

From the Chain Rule of derivative, one can have :

$$\nabla_{W_1} f(W_1, W_2) = \nabla_{Z_2}[\frac{1}{N}\|Z_2 - Y\|_2^2]\nabla_{Z_1} Z_2 \nabla_{W_1} Z_1 + 2\lambda_1 W_1$$

$$\nabla_{W_1} f(W_1, W_2) = \nabla_{Z_2}[\frac{1}{N}\|Z_2 - Y\|_2^2]\nabla_{Z_1}\sigma(W_2\sigma(W_1X^T))\nabla_{W_1}\sigma(W_1X^T) + 2\lambda_1 W_1$$

Hence :

$$\boxed{\nabla_{W_1} f(W_1, W_2) = \frac{2}{N} W_2^T[(\sigma(W_2\sigma(W_1X^T)) - Y) \circ \nabla\sigma(W_2\sigma(W_1X^T)) \circ \nabla\sigma(W_1X^T)]X^T + 2\lambda_1 W_1}$$

The implementation of this is defined as follow :

```
def computePartialDerivativeCLF(X, Y, W1, W2, lambda1, lambda2) :

    W1_XT = np.dot(W1, X.T)
    sig_W1_XT = activation(W1_XT)

    W2_sig_W1_XT = np.dot(W2, sig_W1_XT)
    sig_W2_sig_W1_XT = activation(W2_sig_W1_XT)

    ffwd = (sig_W2_sig_W1_XT - Y.T)

    # for grad_1
    tmp_1 = np.dot(sig_W1_XT, X)
    W2T_ffwd = np.dot(W2.T, ffwd)

    derv_1 = derivateActivation(sig_W2_sig_W1_XT)
    derv_2 = derivateActivation(sig_W1_XT)

    mult_1 = np.multiply(W2T_ffwd,derv_1)
    mult_2 = np.multiply(mult_1,derv_2)

    all_int_mul_1 = np.dot(mult_2, X)

    # for grad 2
    mult_3 = np.multiply(ffwd,derv_1)
    all_int_mul_2 = np.dot(mult_3, sig_W1_XT.T)


    # compute the final result

    grad_1 = (2/X.shape[0])*all_int_mul_1 + 2*lambda1*W1
    grad_2 = (2/X.shape[0])*all_int_mul_2 + 2*lambda2*W2

    return grad_1, grad_2


def computeLoss(X, Y, W1, W2, lambda1, lambda2) :

    W1_W2 = np.dot(W2, W1)
    ffwd = np.dot(W1_W2, X) - Y

    norm_1 =  (1/X.shape[0])*np.linalg.norm(ffwd)
    norm_2 = lambda1*np.linalg.norm(W1)
    norm_3 = lambda2*np.linalg.norm(W2)

    return norm_1 + norm_2 + norm_3
```

1. Same questions from Part I-a) to Part I-f)

a) Back-Propagation with constant step-size

As stated in the remark, the equations for weights update remains the same and adapts to the new gradient formulas. The implementation of Back-Propagation with logistic activation can be defined as follow :

```
def backPropagationCLF(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, alpha, N, with_print) :

    lossArray = np.zeros(N)

    # just to avoid prints when we do cross validation
    if (with_print):
        for i in tqdm(range(N)) :

            loss = computeLoss(X_test, y_test, W1, W2, lambda1, lambda2)

            lossArray[i] = loss

            grad_1, grad_2 = computePartialDerivativeCLF(X_train.T, y_train, W1, W2, lambda1, lambda2)

            W1 = W1 - alpha*grad_1
            W2 = W2 - alpha*grad_2

        print("W2.shape : ", W2.shape)
        print("W1.shape : ", W1.shape)
        print("X.shape : ", X.shape)
        print("Loss mean : ", lossArray.mean())


        return W1, W2, lossArray
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

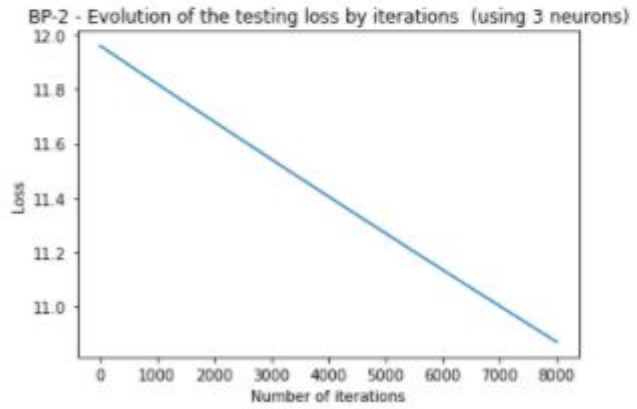
- $\lambda_1 = 0.002$
- $\lambda_2 = 0.1$
- $\alpha = 2e - 05$
- Number of Neurons = 3

```
l1_BP_2, l2_BP_2, alpha_BP_2, n_neuron_BP_2 = getBestParams(lossCV_BP_2, best_valCV_BP_2)
```

```
len(lossTestingKFold) =  3200
len(best_val) =  3200
Best lambda1 :  0.002
Best lambda2 :  0.1
Best alpha :  2e-05
Best n_neuron :  3
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 3.18
- Testing loss : 1.24
- Training MSE : 27.45
- Testing MSE : 3.67




```
The training MSE is : 1185.2806817216215
The testing MSE is : 130.26158558817983
```

b) Mini-Bath SGD

Following the same reasoning as before, the implementation of mini-bath SGD with logistic activation can be defined as follow :

```python
def miniBatchSGD_CLF(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, alpha, N, batch_size, with_print) :

    lossArray = np.zeros(N)

    # just to avoid prints when we do cross validation
    if(with_print) :

        for i in tqdm(range(N)) :

            mini_batches_train = create_mini_batches(X_train, y_train, batch_size)

            for mini_batch in mini_batches_train:

                X_train, y_train = mini_batch
                loss = computeLoss(X_test.T, y_test, W1, W2, lambda1, lambda2)

                lossArray[i] = loss

                grad_1, grad_2 = computePartialDerivativeCLF(X_train, y_train, W1, W2, lambda1, lambda2)

                W1 = W1 - alpha*grad_1
                W2 = W2 - alpha*grad_2

        print("W2.shape : ", W2.shape)
        print("W1.shape : ", W1.shape)
        print("X.shape : ", X.shape)
        print("Loss mean : ", lossArray.mean())
        print("Last loss value : ", lossArray[-1])

        return W1, W2, lossArray
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

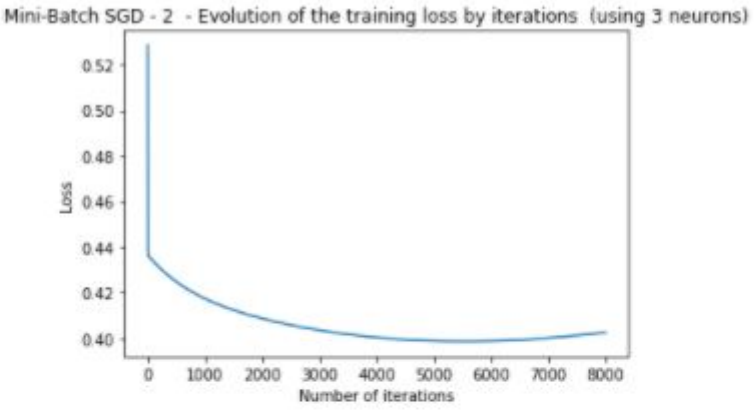
- $\lambda_1 = 0.02$
- $\lambda_2 = 0.08$
- $\alpha = 0.002$
- Number of Neurons = 3

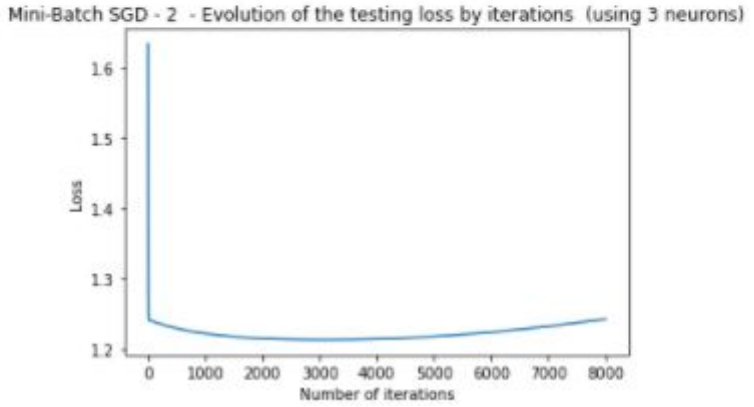
```
l1_SGD_2, l2_SGD_2, alpha_SGD_2, n_neuron_SGD_2 = getBestParams(lossCV_SGD_2, best_valCV_SGD_2)
```

```
len(lossTestingKFold) =  3200
len(best_val) =  3200
Best lambda1 :  0.02
Best lambda2 :  0.08
Best alpha :  0.002
Best n_neuron :  3
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 0.40
- Testing loss : 1.24
- Training MSE : 0.16
- Testing MSE : 1.5






```
The training MSE is : 0.16464991980404015
The testing MSE is : 1.4923315749713433
```

c) ADAM

Following the same reasoning as before, the implementation of ADAM with logistic activation can be defined as follow :

```python
def ADAM_CLF(X_train, X_test, y_train, y_test, W1, W2, lambda1, lambda2, beta1, beta2, n_neuron, N, d, use_one_rs, with_print) :

    lossArray = np.zeros(N)

    # For W1
    U1 = np.zeros((n_neuron,d))
    V1 = np.zeros((n_neuron,d))

    # For W2
    U2 = np.zeros((1,n_neuron))
    V2 = np.zeros((1,n_neuron))

    # delta to prevent any division by zero
    delta = 1e-5

    # just to avoid prints when we do cross validation
    if (with_print):

        for i in tqdm(range(N)) :

            # Compute Loss on prediction
            loss = computeLoss(X_test.T, y_test, W1, W2, lambda1, lambda2)
            lossArray[i] = loss

            if (use_one_rs):

                # compute gradient for one random sample i
                random_sample_indice = np.random.randint(X_train.shape[0])
                random_X = X_train[random_sample_indice].reshape(-1,1)
                random_y = y_train[random_sample_indice].reshape(-1,1)

                grad_1, grad_2 = computePartialDerivativeCLF(random_X.T, random_y, W1, W2, lambda1, lambda2)

            else :
                # compute gradient for X_train
                grad_1, grad_2 = computePartialDerivativeCLF(X_train.T, y_train, W1, W2, lambda1, lambda2)

            # approximate bias for FoM
            U1 = beta1*U1 + (1-beta1)*grad_1
            U2 = beta1*U2 + (1-beta1)*grad_2

            # approximate bias for SoM
            V1 = beta2*V1 + (1-beta2)*grad_1*grad_1
            V2 = beta2*V2 + (1-beta2)*grad_2*grad_2


            # compute bias term of FoM
            x1 = 1- (beta1**i) + delta
            bias_U1 = U1/x1
            bias_U2 = U2/x1

            #compute bias term of SoM
            x2 = 1- (beta2**i) + delta
            bias_V1 = V1/x2
            bias_V2 = V2/x2

            # update weights
            inverse_1 = np.linalg.pinv((np.sqrt(bias_V1) + delta))
            inverse_2 = np.linalg.pinv((np.sqrt(bias_V2) + delta))

            W1 = W1 - inverse_1.T*bias_U1
            W2 = W2 - (inverse_2*bias_U2.T).T

            # to avoid the blowing weight effect
            minmax = MinMaxScaler()
            W1 = minmax.fit_transform(W1)
            minmax = MinMaxScaler()
            W2 = minmax.fit_transform(W2)

        print("W2.shape : ", W2.shape)
        print("W1.shape : ", W1.shape)
        print("X.shape : ", X.shape)
        print("Loss mean : ", lossArray.mean())

        return W1, W2, lossArray
```

In order to find the best hyper-parameters, λ_1, λ_2, and the step-size α, we used a KFold cross-validation with $K = 5$. This lead us to the following values :

- $\lambda_1 = 0.002$

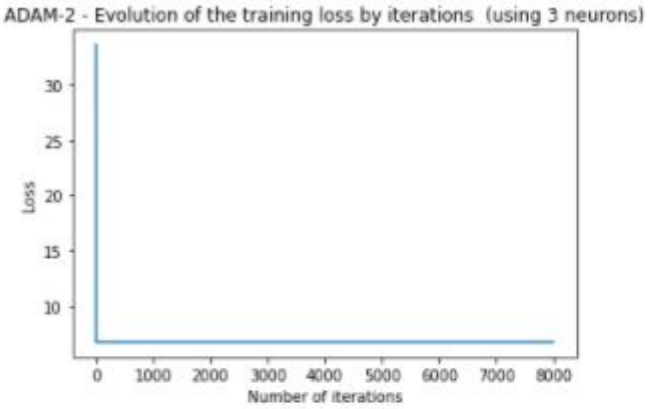
- $\lambda_2 = 0.004$
- $\beta_1 = 0.1$
- $\beta_2 = 0.1$
- Number of Neurons = 3

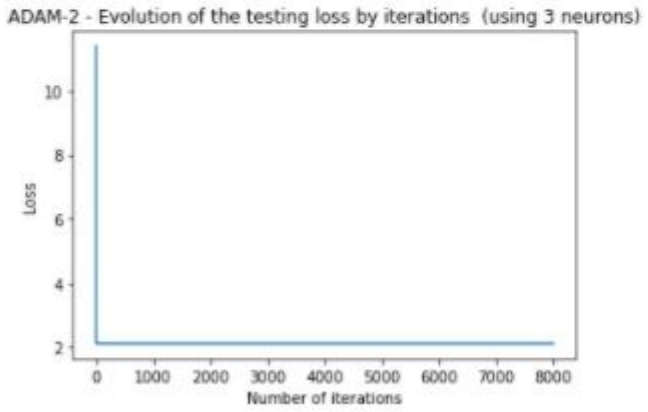
```
l1_ADAM_2, l2_ADAM_2, beta1_ADAM_2, beta2_ADAM_2, n_neuron_ADAM_2 = getBestParams(lossCV_ADAM_2, best_valCV_ADAM_2)
```

```
len(lossTestingKFold) =  5760
len(best_val) =  5760
Best lambda1 :  0.002
Best lambda2 :  0.04
Best beta 1 :  0.95
Best beta 2 :  0.005
Best n_neuron :  3
```

With those values, and after running the algorithm for $N = 8000$ iterations, we achieve :

- Training loss : 6.72
- Testing loss : 2.11
- Training MSE : 45.30
- Testing MSE : 4.47






```
The training MSE is : 45.30939598498211
The testing MSE is : 4.477073977378709
```

d) BCD

In this part, we added a sigmoid activation function in the output of each layer, hence, we can't permute the matrix multiplication. Therefore the trick used in Part I in order to find a close form solution cannot be used here, which **limitates the BCD**.

Furthermore, one can remark that in this case, BCD is like a the Back-Propagation algorithm for each weight W_i.

This can be achieved by the following algorithm :

```
def BCD_First_Component_CLF(X, Y, alpha, W1, W2):

    X_W1T = np.vectorize(activation)(np.dot(X,np.transpose(W1)))
    X_W1T_W2T = np.vectorize(activation)(np.dot(X_W1T,np.transpose(W2)))

    grad_1 = np.multiply(2,np.dot(np.dot(X.T,(X_W1T_W2T - Y)),W2) + np.transpose(W1))
    W1 = W1 - np.multiply(alpha,grad_1).T

    loss = np.multiply(sum(X_W1T_W2T - Y), sum(X_W1T_W2T - Y))

    return W1, loss
```

```
def BCD_SECOND_Component_CLF(X, Y, alpha, W1, W2):

    X_W1T = np.vectorize(activation)(np.dot(X,np.transpose(W1)))
    X_W1T_W2T = np.vectorize(activation)(np.dot(X_W1T,np.transpose(W2)))

    grad_2 = np.dot(W1,np.dot(X.T,(X_W1T_W2T - Y)))
    W2 = W2 - np.multiply(alpha,grad_2).T

    loss = np.multiply(sum(X_W1T_W2T - Y), sum(X_W1T_W2T - Y))

    return W2, loss
```

```
def BCD_CLF(X, Y, W1, W2, N, alpha, start_W1, in_test) :

    lossTest = []
    lossTrain = []
    iterations = [i for i in range(2*N)]

    ts = 0.25

    X_train, X_test, y_train, y_test = train_test_split(X.T, Y, test_size=ts, random_state=42)

    if(start_W1):

        # Optimize W1 then W2
        for i in range(N):
            W1 ,loss_ = BCD_First_Component_CLF(X_train, y_train, alpha, W1, W2)
            lossTrain.append(loss_)
            tmp, loss = BCD_First_Component_CLF(X_test, y_test, alpha, W1, W2)
            lossTest.append(loss)

        for i in range(N):
            W2 ,loss_ = BCD_SECOND_Component_CLF(X_train, y_train, alpha, W1, W2)
            lossTrain.append(loss_)
            tmp, loss = BCD_SECOND_Component_CLF(X_test, y_test, alpha, W1, W2)
            lossTest.append(loss)

    else :

        # Optimize W2 then W1
        for i in range(N):
            W2 ,loss_ = BCD_SECOND_Component_CLF(X_train, y_train, alpha, W1, W2)
            lossTrain.append(loss_)
            tmp, loss = BCD_SECOND_Component_CLF(X_test, y_test, alpha, W1, W2)
            lossTest.append(loss)

        for i in range(N):
            W1 ,loss_ = BCD_First_Component_CLF(X_train, y_train, alpha, W1, W2)
            lossTrain.append(loss_)
            tmp, loss = BCD_First_Component_CLF(X_test, y_test, alpha, W1, W2)
            lossTest.append(loss)

    lossTrain = np.array(lossTrain)
    lossTest = np.array(lossTest)


    MSE_TRAIN = MSE(lossTrain)
    MSE_TEST = MSE(lossTest)

    if in_test == False :

        plt.plot(iterations,lossTest)
        plt.ylabel('Loss')
        plt.xlabel('Number of iterations')
        plt.show()

        print("MSE in TRAIN : ", MSE_TRAIN)
        print("MSE in TEST : ", MSE_TEST)

        return W1, W2, lossTest, lossTrain, MSE_TRAIN, MSE_TEST

    else :
        print("MSE in TEST : ", MSE_TEST)
```

We choose :

- α = 0.0001
- Number of Neurons = 10

With those values, and after running the algorithm for N = 8000 iterations, we achieve :

- Training loss : 0.93
- Testing loss : 0.07
- Training MSE : 90801.57

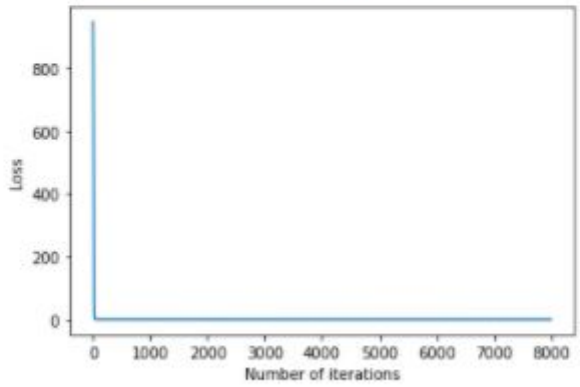
- Testing MSE : 899

Training the BCD

```
alpha = 0.0001
N = 4000
n_neuron = 10
W1 = np.random.rand(n_neuron,d)
W2 = np.random.rand(1,n_neuron)

start_W1 = False
in_test = False
```

```
W1, W2, lossTest, lossTrain, MSE_TRAIN, MSE_TEST = BCD_CLF(X.T, Y, W1, W2, N, alpha,start_W1, in_test)
```



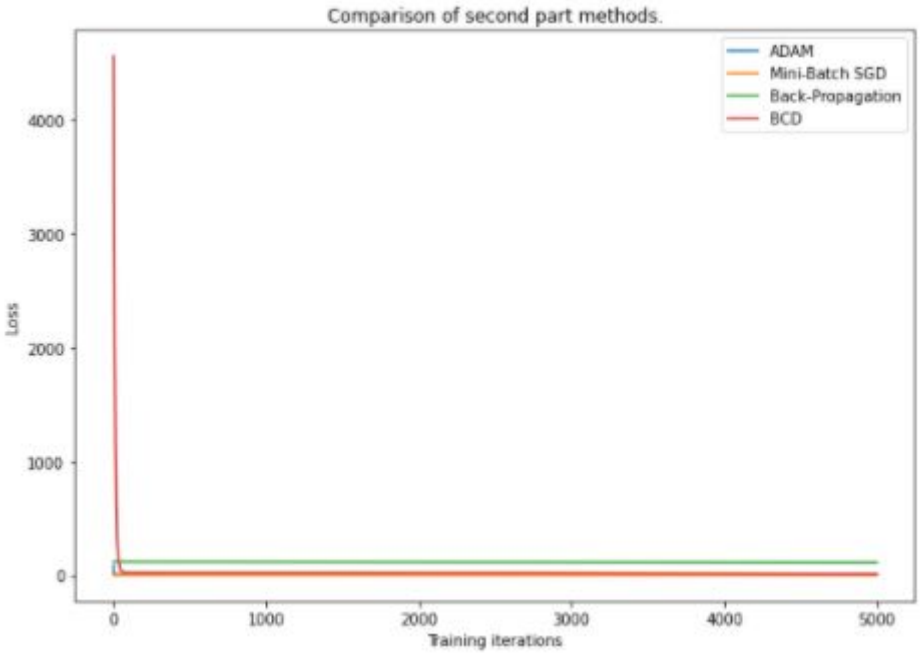

```
MSE in TRAIN :  90801.57209239044
MSE in TEST :  899.0182628133814
```

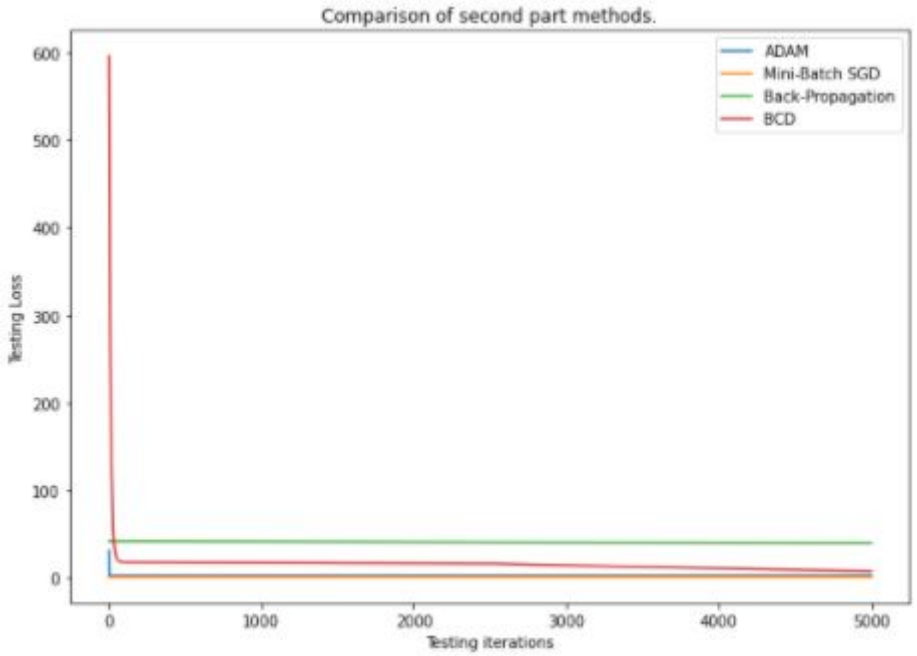
Testing the BCD

```
N = 1
in_test = True
BCD_CLF(X.T, Y, W1, W2, N, alpha,start_W1, in_test)
```

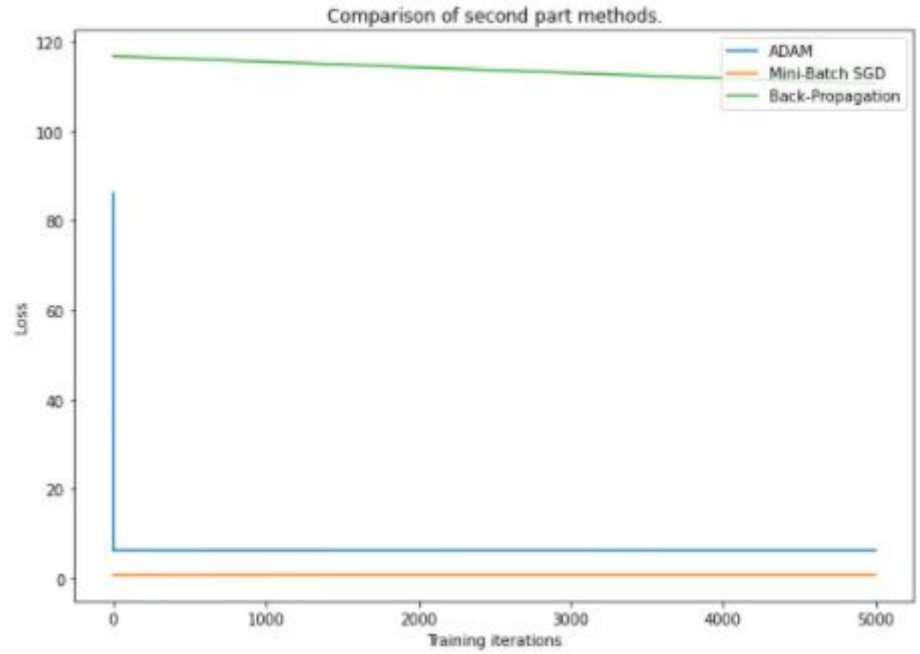
```
MSE in TEST :  0.0053468964203134545
```

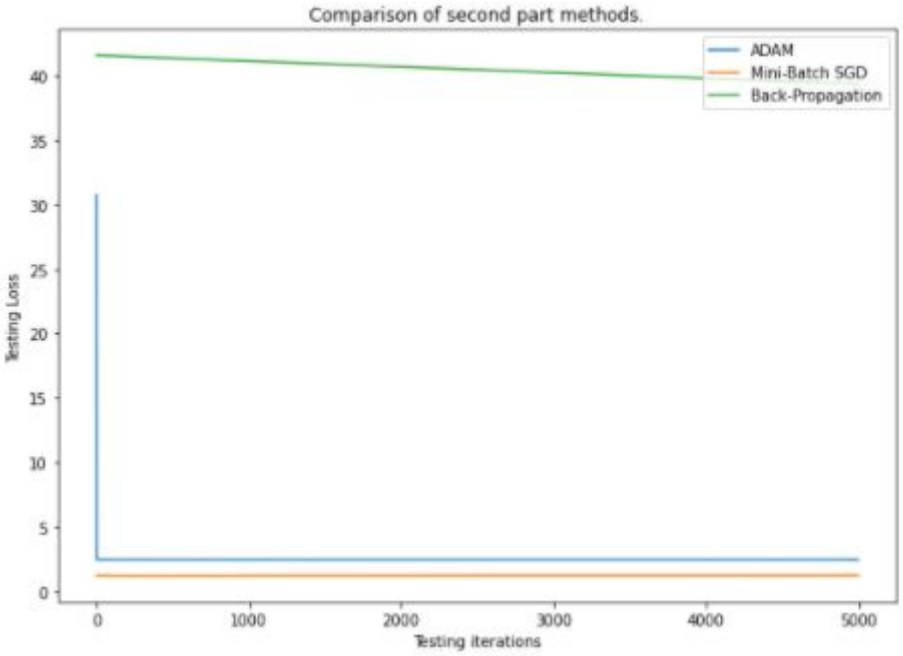
e) Training and test error comparison after convergence is reached





For better Visualization, we can consider those two figure in order to compare the Back-Propagation, Mini-Batch SGD and ADAM methods.





Comments on results

We can clearly see now that the Back-Propagation method didn't perform really well this time. From another point of view, we have that Mini-Batch SGD achieves better testing loss and MSE as compared to ADAM, nevertheless, ADAM shows better performance on MSE when we validated using the **KFold Cross Validation**.

We can resume all the results in the table bellow :

	Training loss	Testing loss	Training MSE	Testing MSE	KFold CV for MSE (K=5)
	Part II				
Back-Propagation	110.42	38.75	12916.97	1638.88	813.72
Mini-Batch SGD	9.12	1.17	0.81	1.47	9.55
ADAM	7.12	3.41	42.08	6.20	3.07
BCD	0.13	0.07	25451.17	682.65	2.15

As the figure of testing loss evolution over the iterations, we concluded that Back-Propagation was the one that has the lowest performance. This statement can be approved by the MSE computed using a **KFold Cross Validation** with $K = 5$ which is really high compared to the others' methods' MSE.

f) Computational complexity Complexity of hyperparameter tuning

The computational complexity in this context depends majorly on the computation of matrix products and matrix inverses. However, in this case it stays the same since our sigmoid activation function do element-wise operations and thus preserves the initial dimensions.

	Computational complexity	Complexity of hyperparameter tuning
Back-Propagation	$O(N \cdot d)$	$O(p^3)$
Mini-Batch SGD	$O(mb \cdot d)$	$O(p^3)$
ADAM	$O(N \cdot d)$	$O(p^4)$
BCD	$O(N^2)$	$O(1)$

2. Training and test error comparison of all the method in Parts I and II

As a result of this project, we finished it by doing a comparison between all the methods using the same number of iterations $N = 5000$ and a number of neurons of 5. We have also used the best hyper-parameters for each model.

These experiments leads us to the table bellow which resumes all the results we have obtained for all the methods we have implemented.

	Training loss	Testing loss	Training MSE	Testing MSE	KFold CV for MSE (K=5)
	Part I				
Back-Propagation	10.43	3.49	1226.88	128.87	4.05
Mini-Batch SGD	10.26	9.62	311.38	393.08	83.78
ADAM	6.54	2.28	48.5	5.61	3.16
BCD	-	-	-	-	-
	Part II				
Back-Propagation	110.42	38.75	12916.97	1638.88	813.72
Mini-Batch SGD	9.12	1.17	0.81	1.47	9.55
ADAM	7.12	3.41	42.08	6.20	3.07
BCD	0.13	0.07	25451.17	682.65	2.15

From these results, one can conclude that using a sigmoid activation can improves the performances of the neural network. In fact the difference between using a sigmoid activation and an identity one should be noticeable since the sigmoid shrinks the value to the interval [0,1] therefore, the distance between it and a normalized/scaled vector of label is smaller and can help in the training by avoiding oscillations around the optimal solution.

The fact that we didn't notice a big difference here in our case is because we have normalized the data in the first part also in order to avoid the blowing weight effect which results in really bad performance.

Nevertheless, there's a noticeable improvement on the Mini-Batch SGD. In fact, the training and testing MSE has shrinked by almost a factor of 400 which i a real gain ! Moreover, the true MSE found by doing the KFold Cross Validation is 9 times better ! So clearly the sigmoid activation helps us with the Mini-Batch SGD, but not with the Back-Propagation for which the performances has gone down drastically.

3 Conclusion

To put in a nutshell, this project was really interesting, it has help me to understand better what's happening under the hood of Neural Networks.

It also made me more aware about the problems that can occur in a Deep Learning project from a mathematical point of view, eg. bad choice of step-size, bad batch-size, blowing weights, etc. Honestly, I think that as a science student, this is really important.