Compte-rendu du TP 3 : Overfitting et surparamétrisation

par Jules Gourio,
et Felix Foucher de Brandois

Formation ModIA - INSA, 4e année
2023-2024

Table des matières

Table des figures

1 Introduction

Le TP overfitting et surparamétrisation représente une ouverture sur une piste de recherche actuelle pour comprendre le fonctionnement des réseaux de neurones. A travers ce compte-rendu, nous allons exprimer les différentes ambitions du TP :

- Améliorer la maîtrise du langage PyTorch.
- Explorer de manière critique le concept d'"overfitting".
- Comprendre le phénomène de double descente et les effets surprenants d'une surparamétrisation.
- Introduire le modèle "random ReLU Feature" (un réseau de neurones simple à une couche).
- Définir une procédure d'entraînement et continuer à explorer la dynamique des algorithmes d'optimisation.
- Comprendre certaines parties du papier de Mikhail Belkin, Fit without fear: remarkable mathematical phenomena of deep learning through the prism of interpolation, Acta Numerica 2021.

2 Random ReLU features

Nous allons ici appliquer un réseau de neurone à une couche très simple appelé random ReLU feature. Il possède la forme suivante :

$$h(x, w) = \sum_{d=1}^{D} w_d \phi_d(x) \quad \text{où} \quad \phi_d(x) = \max(x - b_d, 0) \tag{1}$$

Nous nous intéresserons à une application en 1D, i.e. où $x \in \mathbb{R}$. Ceci permettra de mieux visualiser certains phénomènes, même si les mêmes observations peuvent être reproduites en dimension arbitraire.
Dans notre cadre, les vecteurs b_d sont tirés indépendamment et uniformément au hasard sur $[-1, 1]$. C'est pour cette raison qu'on emploie le terme "random" feature.

1. La figure 1 présente le tracé de la fonction $\phi(\cdot)$ pour différentes valeurs de $b_d \in \mathbb{R}$.

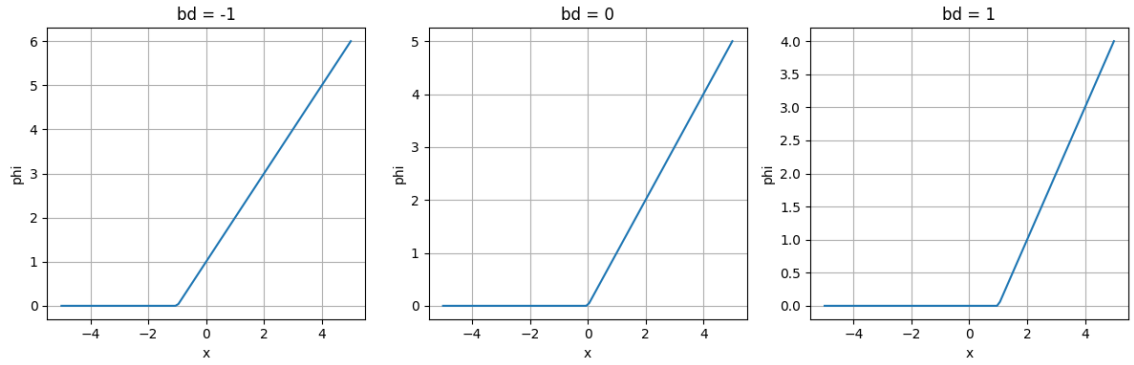

Figure 1: Tracé de la fonction $\phi(\cdot)$ pour différentes valeurs de b_d.

Il s'agit d'une fonction d'activation ReLU.

2. La fonction h est une fonction d'activation ReLU appliquée à une combinaison linéaire, c'est donc une fonction affine par morceaux.
   Les w_k représentent les coefficients directeurs de toutes les droites affines qui permettront d'approximer la fonction recherchée.

   Il est nécessaire d'avoir un grand nombre de ces droites (nombre D) pour obtenir une approximation aussi précise que possible.

3. On suppose que $x \in \mathbb{R}^P, P \in \mathbb{N}$ et que $\phi_d(x) = max(\langle x, v_d \rangle - b_d, 0)$, où v_d vit sur la sphère unité. On a donc : $\phi_d(x) = max(\sum_{i=1}^{P} x_i v_{d,i} - b_d, 0)$

3 Mise en place sur PyTorch

Dans ce TP, nous allons complètement nous affranchir de numpy. C'est une librairie de calcul pratique et plutôt efficace. Cependant, PyTorch offre des fonctionnalités similaires et plusieurs de ses fonctions sont mieux optimisées. De plus, cette librairie permet de travailler sur GPU ou CPU sans effort.

Nous définissons aussi la fonction f à apprendre sur l'intervalle $[-1, 1]$:

```
# The function to learn
def f(X):
    return (torch.abs(X)*torch.sin(2*2*torch.pi*X)).type(dtype)
```

3.1 Préliminaires

1. La figure 2 présente le tracé de la fonction f.

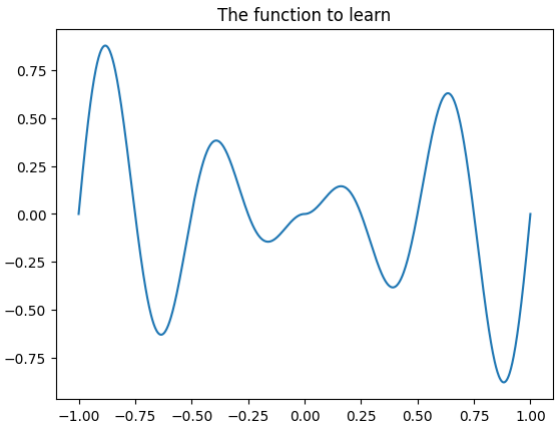

Figure 2: Tracé de la fonction f.

2. On génère un jeu d'apprentissage (x_n, y_n) en échantillonnant la fonction f suivant les positions $x_n = -1 + \frac{2n}{N}$ où N est le nombre total de mesures d'entraînement et $y_n = f(x_n)$.

```
# Number of training samples
N = 100

# Generate training data
X_train = torch.linspace(-1, 1, N).to(device)
Y_train = f(X_train)
```

L'objectif ici va être d'approcher la fonction f à partir des points d'échantillonnage (x_n, y_n) :

Figure 3: Tracé de la fonction f échantillonnée.

Considérons le problème d'optimisation du risque empirique suivant :

$$\inf_{w \in \mathbb{R}^D} R_N(w) \quad \text{avec} \quad R_N(w) = \frac{1}{2N} \sum_{n=1}^{N} ||h(x_n, w) - y_n||_2^2 \tag{2}$$

3.2 Un peu de géométrie

1. On détermine une condition sur les coefficients b_d pour qu'il existe un poids w tel que $R_N(w) = 0$:
   On remarque que la fonction $R_n(w)$ est forcement nulle en -1. Ainsi, pour pouvoir atteindre la premiere valeur du maillage, il faut qu'il existe un b_d avant d'ou : $\exists b_d$ tel que $b_d < x_0$.
   Ensuite, pour atteindre chaque point il suffit d'avoir un b_d entre chaque point du maillage.
   Ainsi, on a : $\forall i \in [1, n], \quad \exists b_d$ tel que $b_d \in]xi - 1, x_i[$.

2. On a : $h(x_n, w) = \sum_{d=1}^{D} w_d \phi_d(x_n)$.

   On pose : $\psi_n = (\phi_1(x_n), \ldots, \phi_D(x_n))$

   On a donc : $h(x_n, w) = \langle w, \psi_n \rangle$

   On pose également : $Y = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \in \mathbb{R}^N$ et $W = \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix} \in \mathbb{R}^D$.

   Soit $\Psi = \begin{pmatrix} \psi_1^T \\ \vdots \\ \psi_N^T \end{pmatrix} \in \mathbb{R}^{N \times D}$.

   On a donc : $R_N(w) = \frac{1}{2N} ||\Psi W - Y||_2^2$

   Le problème (2) devient donc : $\inf_{w \in \mathbb{R}^D} \frac{1}{2N} ||\Psi W - Y||_2^2$

   C'est un problème de moindres carrés : il est convexe et différentiable. La solution est unique si et seulement si Ψ est de rang plein (ou $\Psi^T \Psi$ est inversible).

3.3 Opérations sur les tenseurs

PyTorch permet de faire des opérations sur des tenseurs de taille différentes. Nous avons donc mis en place des expériences en comprendre les méchanismes.

1. On construit un tenseur `A` de taille 3×1 et un tenseur `B` de taille 1×4 :

```
A = torch.tensor([[1], [2], [3]])
B = torch.tensor([[4, 5, 6, 7]])
```

2. On calcule `A + B` en PyTorch :

```
C = A + B
print(C)
# tensor([[5, 6, 7, 8],
# [6, 7, 8, 9],
# [7, 8, 9, 10]])
```

L'opération effectuée ici est une addition terme à terme entre la matrice `A` dont les lignes sont dupliquées et la matrice `B` dont les colonnes sont dupliquées.

3. On construit maintenant un tenseur `A` de taille 3 et un tenseur `B` de taille 4 et on calcule `A[:, N one] + B[N one, :]` :

```
A = torch.tensor([1, 2, 3])
B = torch.tensor([4, 5, 6, 7])
C = A[:, None] + B[None, :]
print(C)
# tensor([[5, 6, 7, 8],
# [6, 7, 8, 9],
# [7, 8, 9, 10]])
```

On obtient le même résultat qu'à la question précédente.

3.4 Le code principal

1. On construit une classe PyTorch définissant le modèle h. Elle prend la forme suivante :

```
class one_layer_NN(nn.Module):
    def ___init___(self, n_hidden = 10):
        super(one_layer_NN, self).___init___()
        self.b = 2*(torch.rand(n_hidden)-0.5).to(device)
        self.w = torch.nn.Parameter(torch.zeros(n_hidden))
    def forward(self, x):
        dif = x - self.b[None,None,:]
        hidden = torch.maximum(dif,torch.zeros_like(dif))
        return torch.sum(self.w[None,None,:]*hidden, keepdim=True,
dim=2)
```

2. En prenant exemple sur le TP2, nous avons codé un algorithme d'optimisation stochastique pour minimiser le risque empirique sous PyTorch.

3. On affiche la fonction $h(\cdot, w)$ toutes les 1000 ainsi que son adéquation aux données sur la figure suivante :

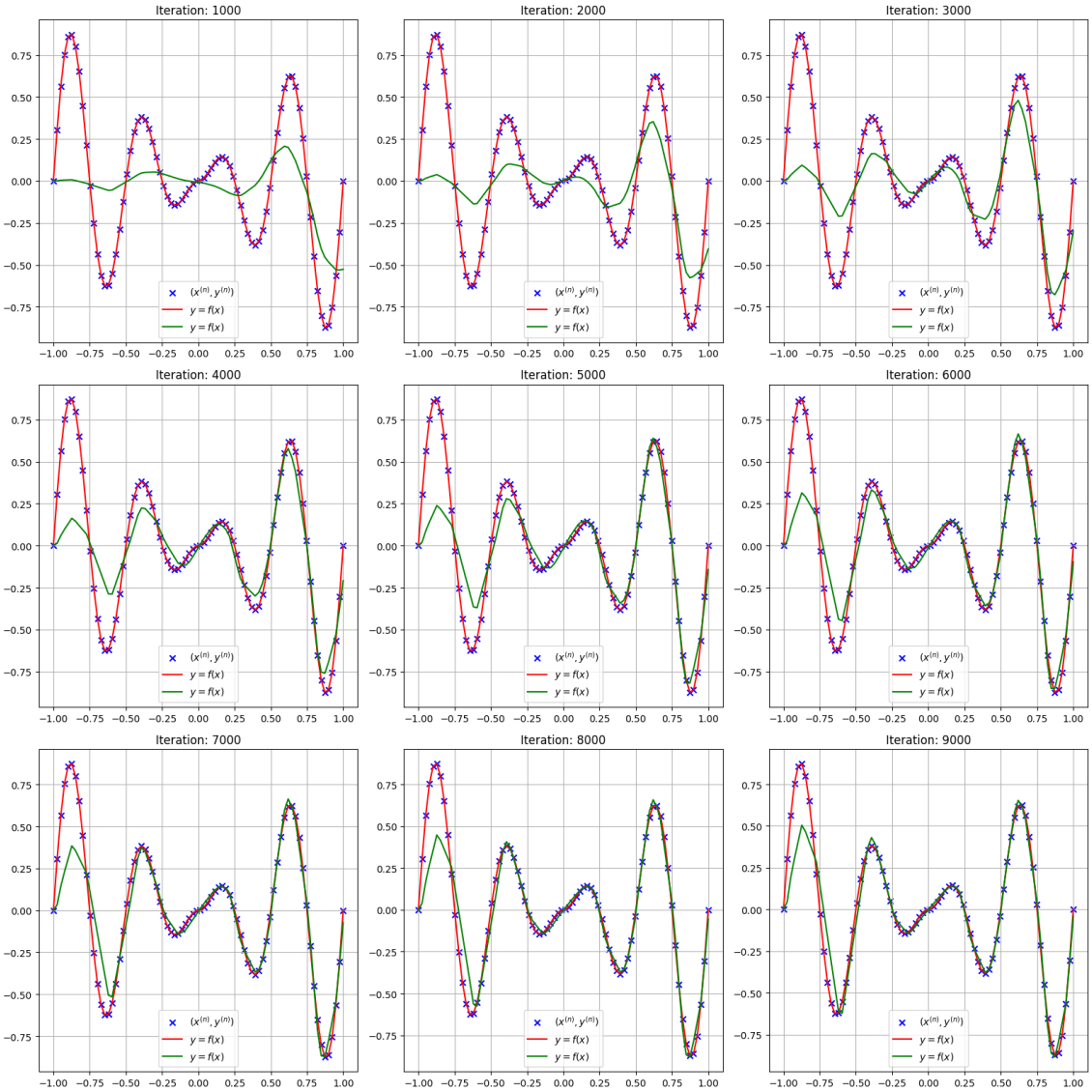

Figure 4: Fonction $h(\cdot, w)$ et son adéquation aux données.

Après avoir effectué plusieurs tests d'optimiseurs (avec un nombre de points $N = 100$ et $D = 100$), on décide de garder l'optimiseur Adam de la librairie torch.optim avec un taux d'apprentissage : $lr = 10^{-3}$.
La figure ci-dessus illustre l'apprentissage de la fonction au cours des itérations avec ces paramètres.

4 Analyse des résultats

1. Pour $N = 10$, on obtient l'échantillonnage suivant :

Figure 5: Tracé de la fonction f échantillonnée avec $N = 10$.

2. On calcule la solution de 2 pour différentes valeurs de D.

   - En régime sous-paramétré :

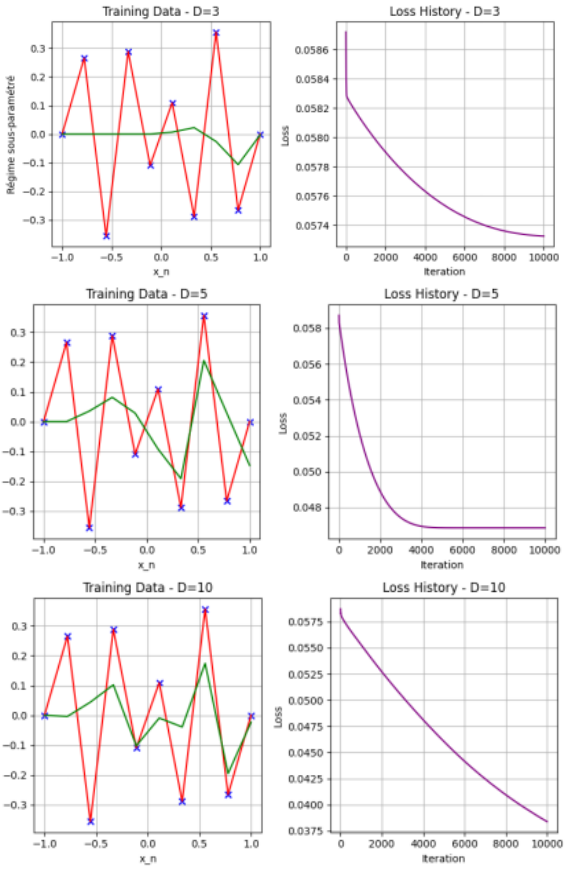

Figure 6: Solution en régime sous-paramétré.

Pour ce régime avec de faibles valeurs de D le résultats obtenu n'est pas très satisfaisant. La fonction verte qui est la fonction prédite est très loin de la fonction originale. De plus on voit sur l'historique de la fonction loss

que ces 3 modèles ne vont pas converger, la fonction perte stagne autour d'une valeur égale à 10^{-2} .

- En régime limite :

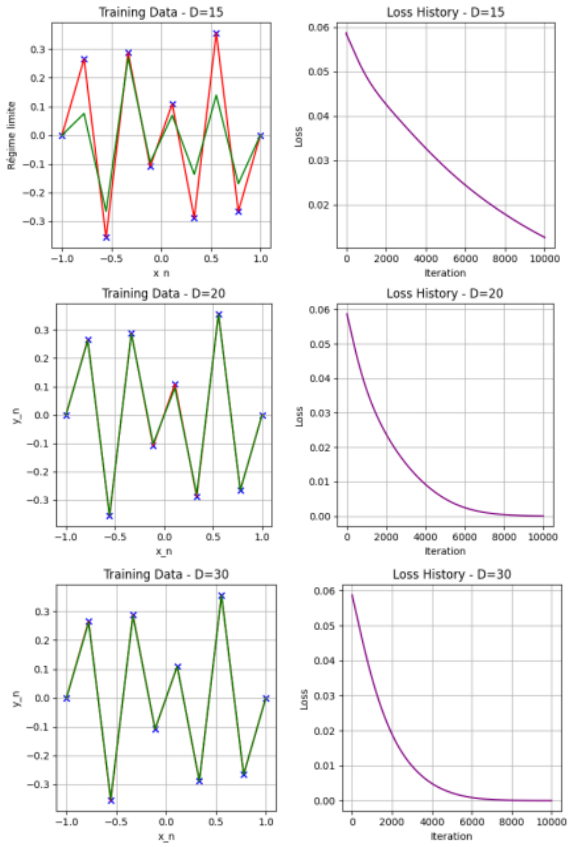

Figure 7: Solution en régime limite.

Les fonctions prédite des modèles en régime limite sont beaucoup plus proches de la fonction à estimer. A partir de $D = 20$ la fonction est quasiment reproduite à l'identique avec des loss très faible de l'ordre de 10^{-5} . Selon la courbe de 10 on voit que l'on a déjà atteint la valeur limite entre la sous-paramétrisation et la sur-paramétrisation.

- En régime sur-paramétré :

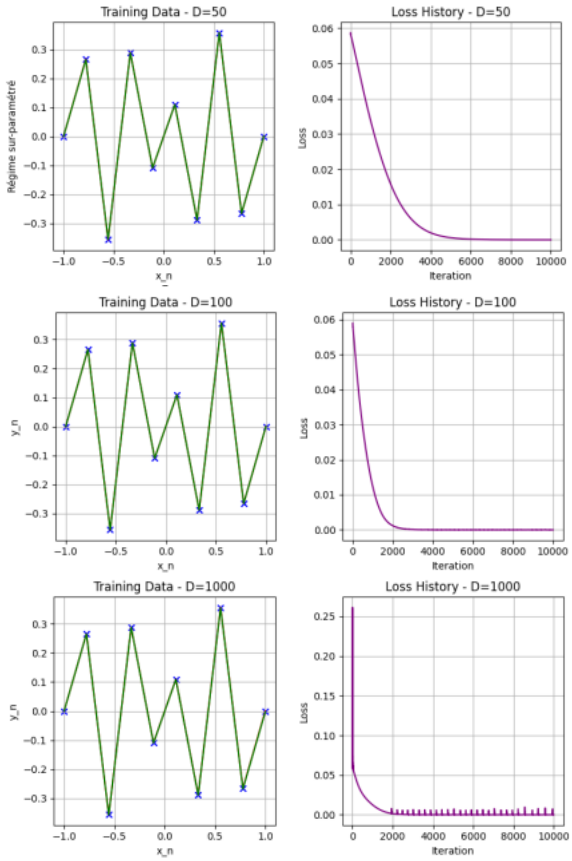

Figure 8: Solution en régime sur-paramétré.

Dans le régime surparamétré les 3 fonctions prédites se surperposent totalement à la fonction à estimer et on remarque que la fonction loss converge très rapidement vers 0. Mais un phénomène très bizarre apparaît, avec $D = 100$ on observe des pics dans la fonction loss qui sont des témoins purs de l'overfitting.

3. Nous allons essayer de quantifier cette observation. Pour ce faire, nous avons tracé le risque moyen pour différentes valeurs de D à la figure 9.

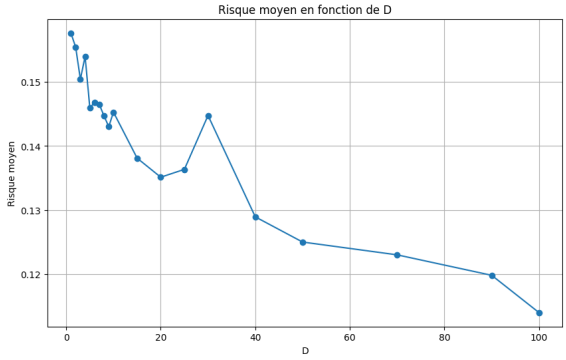

Figure 9: Risque moyen pour différentes valeurs de D.

L'allure de la courbe est cohérente avec la courbe de référence :

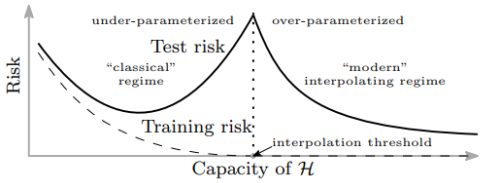

Figure 10: Courbe de référence.

5 Gradient stochastique et surparamétrisation

Finalement, on se propose d'analyser empiriquement le comportement du gradient stochastique dans le régime surparamétré. Da façon informelle, on a le résultat suivant :

> Dans un régime surparamétré, le gradient stochastique à pas constant converge, tandis qu'il ne converge pas sinon.

La raison intuitive derrière cette propriété est que les gradients des fonctions individuelles f_i s'annulent de façon synchrone sur les minimiseurs globaux. C'est un peu comme s'il y avait une réduction de variance automatique.

1. Nous avons lancé une descente de gradient stochastique avec un mini-batch pour $D = 5$ et $D = 1000$. De plus nous avons configurer l'algorithme de gradient stochastique avec un nombre de points N = 100 et un $batch\_size = 10$.

   La figure 11 présente le risque empirique à chaque itération pour ces deux valeurs de D.

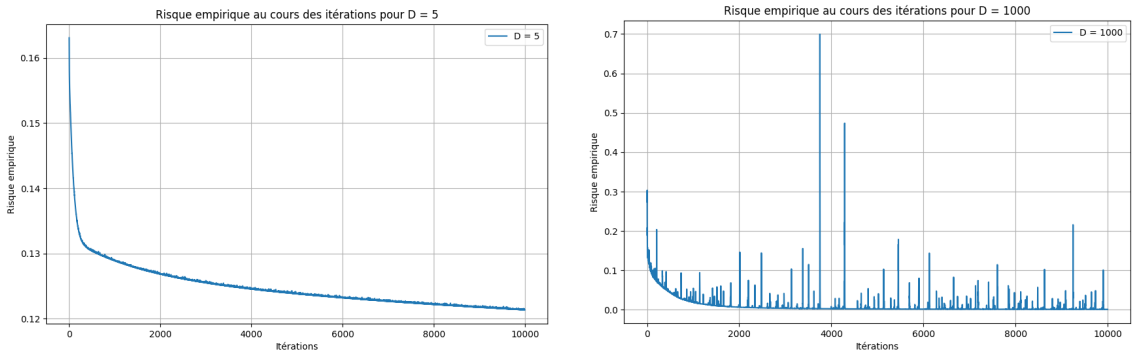

Figure 11: Risque empirique à chaque itération pour $D = 5$ et $D = 1000$.

2. Est-ce que vous voyez une différence de comportement dans le régime sur-paramétré et dans le régime sous-paramétré ?

   On remarque sur ces 2 figures les différences entre les le régime sous-paramétré et le régime sur-paramétré. Le régime sous-paramétré à un très gros défaut, celui-ci n'arrive pas à converger, le risque empirique stagne même en augmentant le nombre d'itérations. Le régime sur-paramétré lui converge mais il y a un autre problème caractéristique de l'overfitting, il y a des énorme pics dans la fonctions loss ainsi en fonction de l'itération il peut donner un risque empirique totalement abberant.