Rapport TME1 : Regression

CELLIER Roxane - MOULIN-ROUSSEL Lou

1

1. Batch Linear Least Squares

Pour étudier les différences entre les deux premières méthodes proposées, nous avons effectué 2000 tests sur différents jeux de données de même taille (50), pour nous permettre d'évaluer les temps d'exécution et les écarts entre les paramètres appris. La boucle que nous avons utilisée est la suivante :

```
    for i in range(2000):
        batch = Batch()
        size = 50
        batch.make_linear_batch_data(size)
        model = LinearModel(size)
        x = np.array(batch.x_data)
        y = np.array(batch.y_data)

        start = time.process_time()
        model.train(x, y)
        theta1[i,:] = model.theta
        time_lls[i] = time.process_time() - start

        start = time.process_time()
        model.train_from_stats(x, y)
        theta2[i, :] = model.theta
        time_lls_stats[i] = time.process_time() - start

```

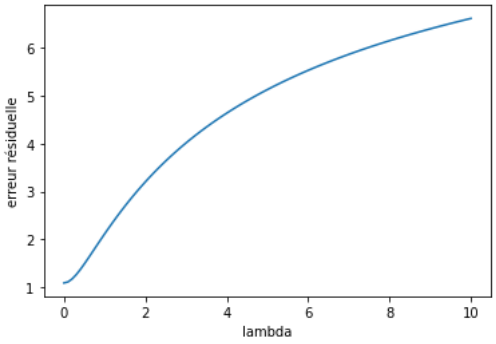
Ce qui nous a permis d'obtenir les résultats suivants en calculant les moyennes :

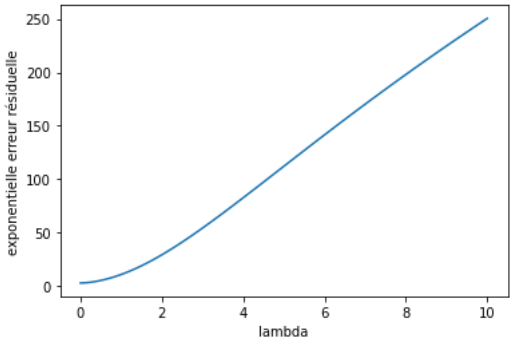
```
différences de theta :  [6.66133815e-16 0.00000000e+00]
moyenne de temps lls :  0.00012791794249999988
moyenne de temps lls avec stats :  0.00046454361200001413
```

On semble constater que l'utilisation de stats est légèrement moins rapide, même si les deux méthodes s'exécutent en un temps du même ordre grandeur. De plus, les valeurs renvoyées des paramètres theta ne sont pas forcément exactement identiques, mais cependant extrêmement proches. Cela peut être dû à des approximations et arrondis faits par la machine lors des calculs.

2. Ridge Regression

Pour ce test, nous avons considéré un jeu unique de taille 50, pour lequel nous avont testé différentes valeurs du paramètre lamba, de 0 à 10 par pas de 0,1, pour ainsi observer la variation de l'erreur.





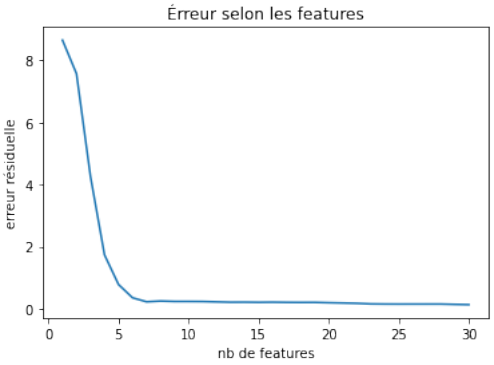
Nous remarquons que l'erreur résiduelle croît lorsque le poids lambda augmente. Elle semble augmenter de manière logarithmique. En effet, sur la deuxième figure ci-dessus, nous pouvons voir que l'exponentielle de l'erreur suit une progression affine voire linéaire en fonction de lambda.

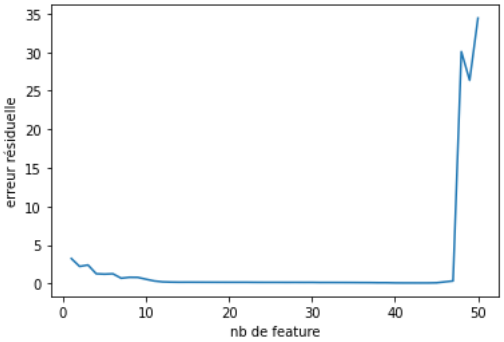
3. Radial Basis Function Networks

De la même manière que dans la première partie (Batch Linear Least Squares), nous avons effectué 2000 tests sur des jeux différents pour pouvoir comparer en moyenne le temps d'exécution des deux approches présentées dans ce TME. Nous constatons assez vite que les deux perspectives ont des temps d'exécution très similaires, la première approche semblant tout de même sensiblement plus rapide que la deuxième.

```
moyenne de temps RBNF1 :  0.0005012852634999945
moyenne de temps RBNF2 :  0.0007240960204000352
```

Nous regardons maintenant l'évolution de l'erreur sur un même jeu de données de taille 60, selon le paramètre lambda. Nous avons choisi ici d'utiliser la première perspective, celle-ci étant sensiblement plus rapide.

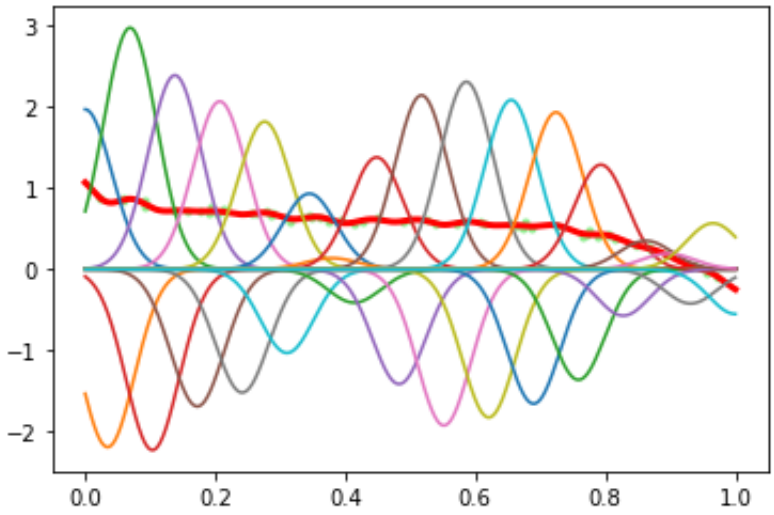




L'erreur résiduelle décroît progressivement plus on augmente le nombre de features. On observe une diminution très forte entre 0 et 5 features et une diminution qui ne semble plus significative au-dessus de 10 features. Cependant sur certains tests, l'écart remonte fortement autour de 50 features, comme illustré par la figure ci-dessus.

Avec 30 features, on obtient le graphe ci-dessous, avec en rouge et en gras la courbe

obtenue par régression et en vert les points avec lesquels la régression est faite. Les autres courbes représentent les features.



Nous constatons ici un overfitting du modèle sur les données de test : la courbe formée est très proche de chacun des points et nécessite de nombreux paramètres pour être décrite.

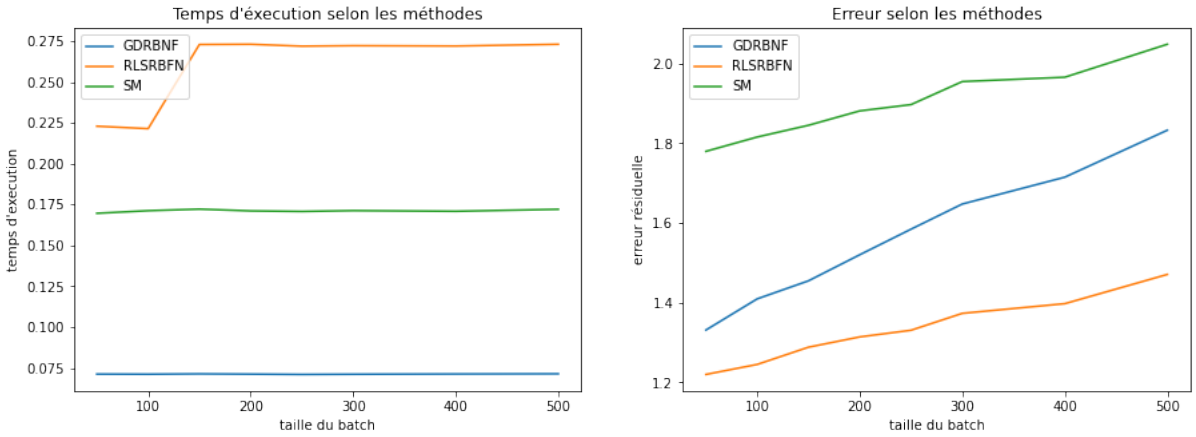
4. Incremental RBFNs

Nous allons ici tenter de comparer globalement chacune des méthodes pour estimer ce qui rend une approche meilleure qu'une autre. Nous avons voulu dans un premier temps constater les éventuelles variations liées à la taille du jeu de données. Nous avons donc pris différentes tailles allant de 50 à 500, sur lesquelles nous avons effectué nos tests sur 1000 jeux différents pour nous permettre de fournir une moyenne. Nous avons choisi pour ces tests de fixer le nombre de feature à 15 et le nombre maximal d'itération à 500. Voici le code effectuant les 1000 tests pour une taille t spécifique :

```
    for i in range(NB_TRY):
    # cr ation des jeux de donn es
    batch = Batch()
    batch.make_nonlinear_batch_data(size=t)
    x = np.array(batch.x_data)
    y = np.array(batch.y_data)

    # test temps GDRBFN
    modelGD = GDRBFN(nb_features=NB_FEATURE)
    start = time.process_time()
    for j in range(MAX_ITER):
        x, y = batch.add_non_linear_sample()
        modelGD.train(x, y, alpha=0.5)
    time_GDRBNF[i] = time.process_time() - start
    error_GDRBNF[i] = modelGD.compute_error(batch.x_data,batch.y_data)

    # test temps RLSRBFN
    modelRLS = RLSRBFN(nb_features=NB_FEATURE)
    start = time.process_time()
```

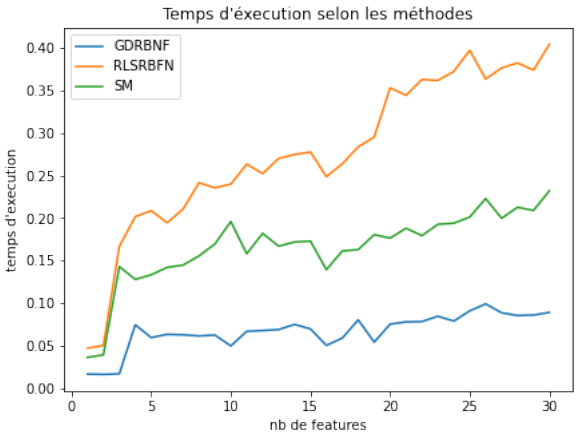
```
    for j in range(MAX_ITER):
        x, y = batch.add_non_linear_sample()
        modelRLS.train(x, y)
    time_RLSRBFN[i] = time.process_time() - start
    error_RLSRBFN[i] = modelRLS.compute_error(batch.x_data,batch.y_data)

    # test temps Shermann - Morrison
    modelSM = RLSRBFN(nb_features=NB_FEATURE)
    start = time.process_time()
    for j in range(MAX_ITER):
        x, y = batch.add_non_linear_sample()
        modelSM.train_rls_sherman_morrison(x, y)
    time_SM[i] = time.process_time() - start
    error_SM[i] = modelSM.compute_error(batch.x_data,batch.y_data)

```

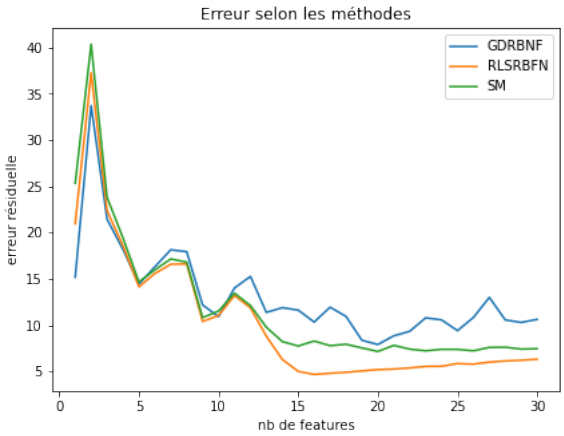
Ce qui nous donne comme affichage :



Les temps d'exécution selon les méthodes semblent constants en fonction de la taille du batch, excepté pour la méthode Recursive Least Squares qui semble avoir un léger sursaut autour d'un jeu de 100 valeurs. Les erreurs selon les méthodes semblent quant à elles augmenter légèrement en fonction de la taille du batch. Cela peut paraître logique puisque la taille des données augmente, mais pas le nombre de features permettant de coller à ces données.

Nous cherchons maintenant à faire varier le nombre de features pour un même jeu de données, puis nous affichons les temps d'exécution et les erreurs selon les méthodes.

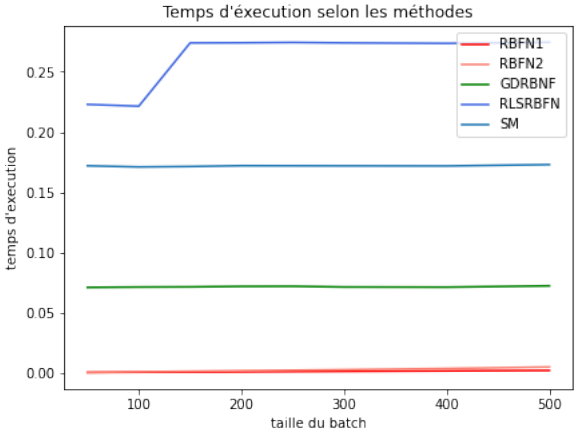


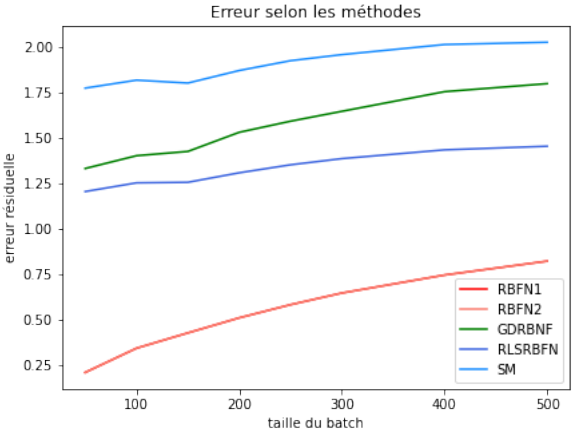


On observe que la méthode de descente de gradient semble plus rapide mais l'erreur est plus grande lorsque le nombre de features est supérieur à 10 environ. La méthode de recursive least squares sans l'approche de Sherman-Morrison (RLSRBFN) est la plus lente mais aussi celle avec la plus petite erreur.

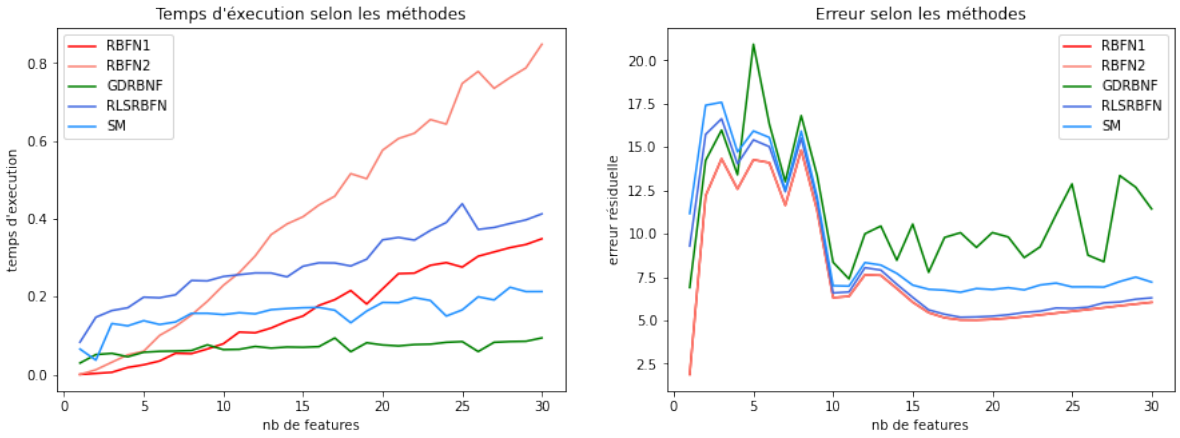
5. Comparaison Batch et Incremental

En utilisant les mêmes types de test et les mêmes paramètres (nombre de tests, tailles des jeux, etc) sur l'ensemble des méthodes RBNF, nous obtenons les mêmes types de tracé. Pour simplifier la visualisation, les méthodes Batch sont teintées en rouge et les méthodes Incremental sont teintées en bleu.





Nous pouvons voir de manière assez nette que les méthodes batch semblent nettement plus rapide que les méthodes incrémetales, elles semblent également renvoyer moins d'erreur que les autres méthodes.



On peut remarquer que pour un faible nombre de features, les techniques incrémentales sont plus lentes et ont une plus large erreur résiduelle. Cependant si l'on augmente grandement le nombre de features, on peut voir que le temps d'exécution croît beaucoup plus vite et en vient même à dépasser celui des méthodes incrémentales. De manière globale, la méthode du gradient descendant semble la plus rapide, en effet son temps d'exécution ne varie presque pas en fonction du nombre de features, mais comme précedemment, son erreur est la plus grande pour un large nombre de features.

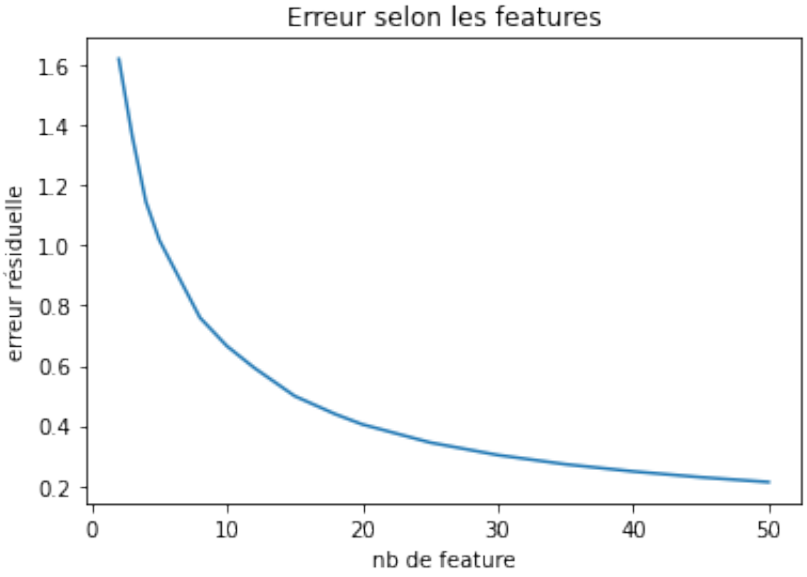
Au vue des courbes visibles, il n'est pas évident de décider qu'une méthode est largement supérieure à toutes les autres. Cela va fortement dépendre de ce que l'on chercher à minimiser (temps ou erreur), de la taille des données, et également du nombre de features que l'on souhaite récupérer pour notre modèle.

6. Locally Weighted Regression

Nous allons ici visualiser l'impact du nombre de sous-divisions sur l'erreur du modèle appris par LWR. Pour cela, nous allons prendre un jeu de données fixe sur lequel nous allons apprendre différents modèles selon le nombre de sous-ensembles locaux voulus. Nous testerons l'erreur pour différents découpages de 2 à 50 ensembles. Le code nous permettant d'effectuer ces tests est présenté ci-dessous :

```
    batch = Batch()
    batch.make_nonlinear_batch_data()

    nb_feature = [2,3,4,5,8,10,12,15,18,20,25,30,35,40,45,50]
    error = np.zeros(len(nb_feature))

    for i in range(len(nb_feature)):
        model = LWR(nb_features=nb_feature[i])
        model.train(batch.x_data, batch.y_data)

        error[i] = model.compute_error(batch.x_data, batch.y_data)

```

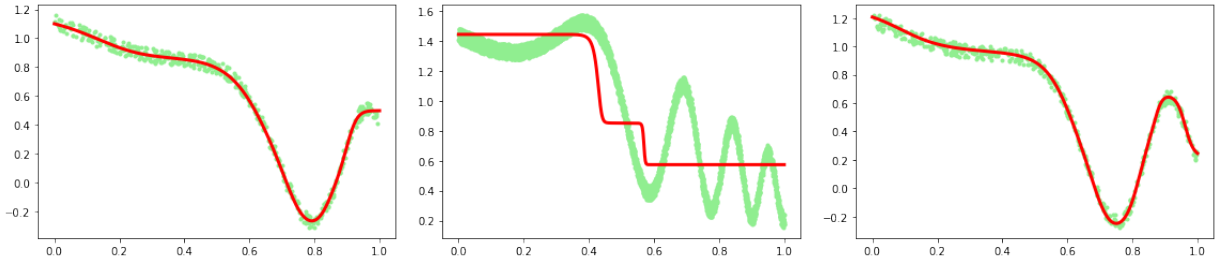
Suite à l'exécution du code et à l'affichage, nous obtenons le tracé suivant :



De façon assez logique, on peut voir que l'erreur baisse au fur et à mesure que l'on divise le jeu en plusieurs sous-ensembles. Il semble en effet plus efficace pour essayer de suivre une courbe de faire plusieurs petits segments plutôt qu'une seule droite.

7. Regression with Neural Networks

Nous allons observer différentes utilisations de PyTorch pour apprendre différents modèles autour de différents jeux, et étudier les différents résultats ainsi que le respect des données par le modèle ainsi appris.



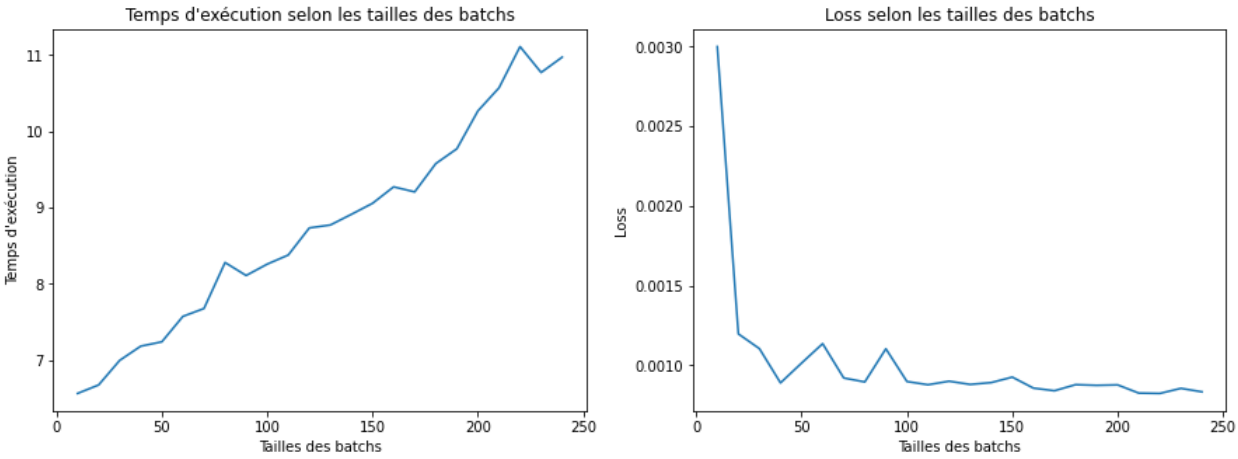
On peut constater que le modèle résultant de la méthode Batch (à droite) et celui résultant de la méthode incrémentale utilisant des mini-batches (à gauche) sont cohérents avec le jeu de données. Cependant la version incrémentale (au milieu) ne donne pas un résultat concluant. Le modèle proposé ressemble plus à une fonction en escalier, et les marches semblent représenter une moyenne générale sur la largeur qu'elles représentent.

Nous allons ensuite étudier l'évolution du temps d'exécution et de la valeur de la loss suivant la dernière méthode, en faisant varier les tailles des mini-batches. Nous avons utiliser cette boucle nout permettant d'effectuer nos tests sur une taille s :

```
    for t in range(nb_try):
        start = time.process_time()
        for i in range(max_iter):
            xt, ym = make_minibatch(minibatch_size, batch.x_data, batch.y_data,
     size)
```

```
            yt = th.from_numpy(ym).float()
            output = model.f(xt)
            loss = func.mse_loss(output, yt)
            #print(loss.item(), model.compute_error(xt,yt))
            model.update(loss)
        #print("NN Incr Reg time:", time.process_time() - start)
        times_s[t] = time.process_time() - start
        loss_s[t] = loss

```

Nous avons utilisé la valeur `nb_try = 10` pour avoir une moyenne sur 10 valeurs pour chaque taille de minibatchs. Ce qui nous permet d'obtenir l'affichage suivant :



On remarque que l'évolution du temps d'exécution en fonction de la taille des minibatchs semble relativement linéaire. Alors que la loss semble décroître de manière très rapide avec des tailles de minibatchs assez petites (jusqu'à environ 50) puis ne plus beaucoup évoluer. On aurait donc tendance à choisir une taille de minibatch autour de 50.