Réseaux connexionnistes

Implémentation d’algorithmes pour l’apprentissage supervisée & application

Décembre 2022, COULON Axel

08/12/2022

Sommaire

RÉSEAUX DE NEURONES SUPERVISÉS

I - Algorithmes d'optimisation

I.a. - Programmation

L'implémentation des algorithmes : Batch Gradient Descent, Stochastic Gradient Descent (SGD), AdaGrad , RMSprop , SGD avec momentum et ADAM sont disponibles dans le fichier DM1_ARI_COULON disponible ci-joint ou sur mon github à cette adresse.

I.b. - Objectif

On souhaite apprendre, avec un perceptron, la relation que l'on suspecte linéaire entre les degrés Celsius (entrée) et les degrés Fahrenheit (sortie). La formule exacte que l'on cherche est $1.8 * C + 32 = F$.

I.c. - Génération des données

Création des données et de la relation F=aC + b + bruit

```
num_samples = 100
X = np.ones((num_samples,2))  #creer une matrice de de 100*2 rempli de 1
X[:,0] = 20*np.random.uniform(-1.,1.,num_samples)+10 # rempli la premiere colonne de valeurs entre -10 et +30 reparti uniformémer
K = 20*np.random.uniform(-1.,1.,num_samples)+10
a = np.array([1.8,32])
y = X.dot(a) + np.random.normal(size=num_samples)  # on retrouve la relation y=1.8x(:,0) + 32*x(:,1) + bruit gaussien
```

I.d. - Application des algorithmes de descente du gradient pour la régression linéaire

La Descente de Gradient est un algorithme d'optimisation qui permet de trouver le minimum de n'importe quelle fonction convexe en convergeant progressivement vers celui-ci.

En Machine Learning, on va utiliser l'algorithme de la Descente de Gradient dans les problèmes d'apprentissage supervisé pour minimiser la fonction coût, qui justement est une fonction convexe (par exemple l'erreur quadratique moyenne (mean square error, qu'on utilise ici). Voici un dessin descriptif montrant l'application de l'algorithme du descente de gradient pour optimiser un paramètre *a* sur la fonction de cout *j(a,b).*

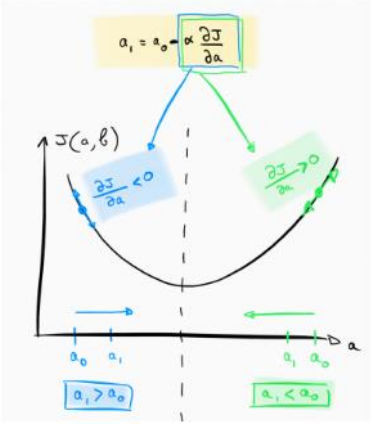

Figure 1 : schema expliquant l'algorithme de descente du gradient

Dans notre cadre applicatif, notre algorithme de descente du gradient et ses dérivés (SGD, SGDM, AdaGrad, RMSProp,ADAM) essayerons de réduire au maximum l'erreur quadratique moyenne entre la fonction $1.8 * C + 32 = F$ et une fonction calculée par l'algorithme y=a0*C +a1*1 avec a0 et a1 nos poids à optimiser.

Dans cette partie on se concentrera à l'analyse des résultats : évolution des poids, fonction de perte, temps de calcul, nombre d'itérations… plutôt qu'à l'implémentation des algorithmes. Vous retrouverez l'implémentation des algorithmes sur mon GitHub ici.

Les données ne sont pas normalisées dans un premier temps car l'ordre de grandeur des données n'est pas différent d'une donnée à une autre.

Arbitrairement, nous instancions les valeurs de nos paramètres a0 et a1 à respectivement 28 et 34.

<u>À préciser : le Nombre d'itérations nécessaires à la convergence sera calculé lorsque les coefficients de la relation degré Celsius, fahrenheit sont quasiment identiques à la vraie relation. Il ne faut pas tenir compte, pour ce paramètre, de la convergence sur les animations.</u>

I.d.i. - Batch Gradient Descent

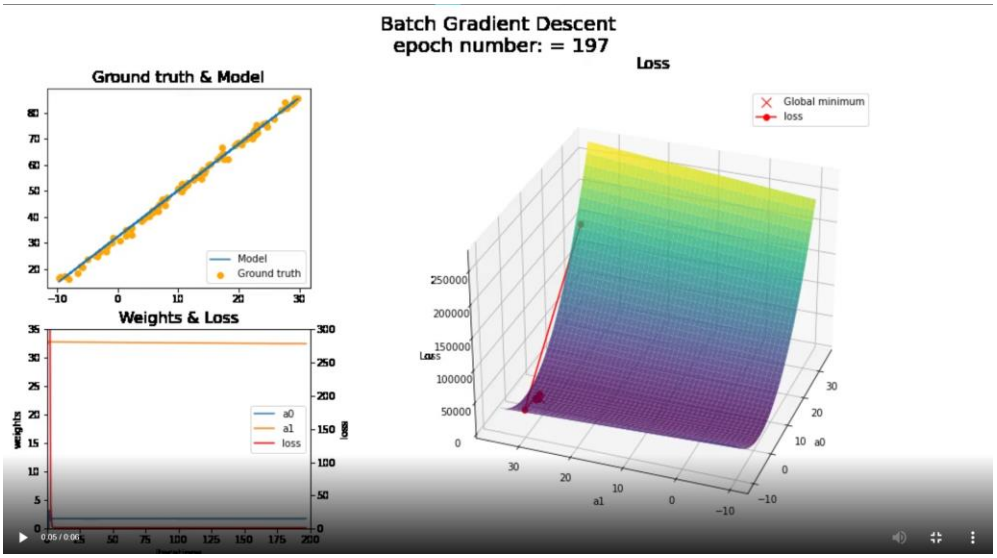

Figure 2 : Animations montrant la performance de l'algorithme Batch Gradient Descent

Paramètres (alpha/epochs)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
0.005 / 200	0.055878	5	Converge très rapidement et stable
0.005 / 10	0.00398	7	Converge très rapidement et stable
0.01/200	0.057556	Pas de convergence	Très instable et diverge
0. 0005 / 200	0.067953	57	Converge assez rapidement et très stable

On remarque que l'algorithme BGD converge extrêmement rapidement en 5 itérations après seulement 0.05 secondes.

I.d.ii. - Stochastic Gradient Descent

Particularité : C'est une descente de gradient mais qui va se focaliser sur des échantillons différents du jeu de données global à chaque itération.

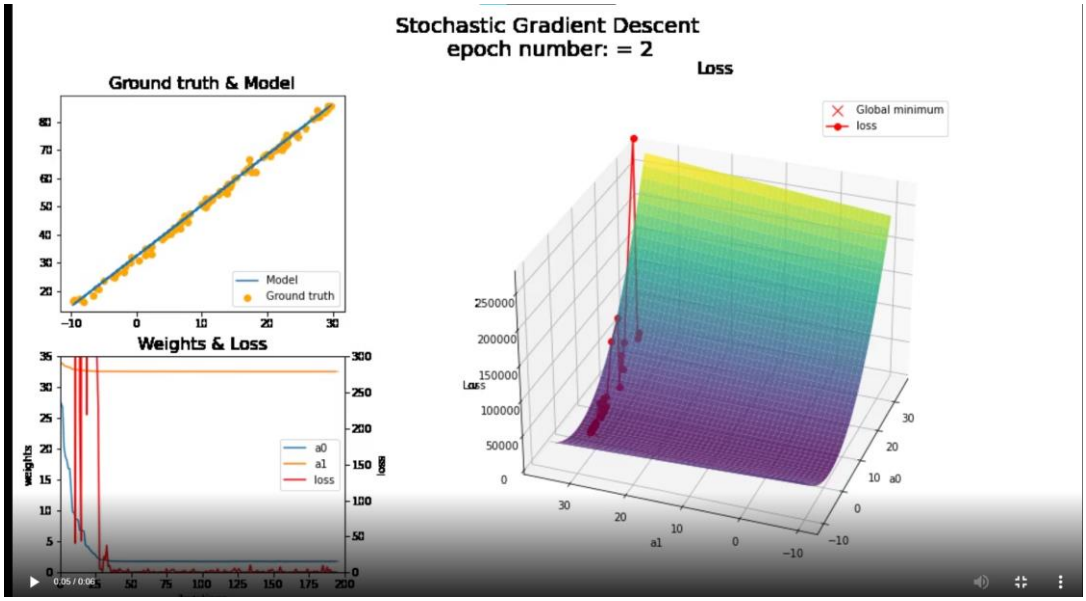

Figure 3 : Animations montrant la performance de l'algorithme Stochastic Gradient Descent

Paramètres (alpha/epochs)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
0.0005 / 200	0.050863	42	Converge rapidement et stable

0.0005 / 100	0.032911	46	Converge rapidement et stable
0.005 / 200	0.045876	40	Rapide mais très instable + saut + pas les mêmes résultats à chaque essais
0.005 / 300	0.067953	162	instable + saut + pas les mêmes résultats à chaque essais

I.d.iii. - AdaGrad

Particularité : Cet algorithme met à l'échelle de manière adaptative le taux d'apprentissage pour chaque dimension.

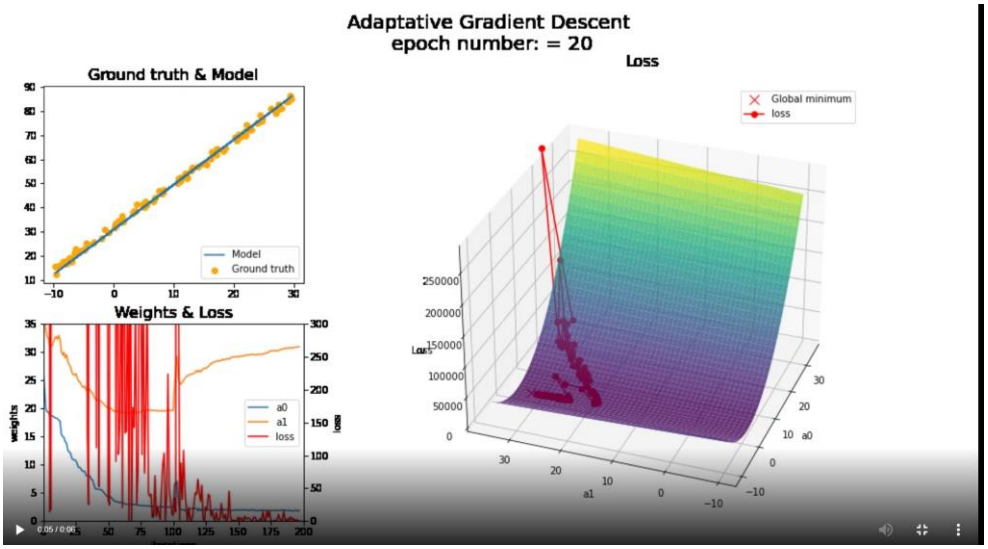

Figure 4 : Animations montrant la performance de l'algorithme AdaGrad

Paramètres (alpha/epochs)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
4/200	0.058863	197 proche du résultat mais nécessite plus d'itérations	Converge lentement et stablement
4/400	0.106082	278	Converge lentement , stable mais saut possible

2/400	0.102734	310	Converge lentement , stable, moins de saut.
0.8/400	0.102735	Converge oui, mais trop lentement, n'arrive pas aux valeurs désirés	Stable mais convergence trop lente

I.d.iv. - RMSProp

Particularité : RMSprop utilise une moyenne mobile des gradients au carré pour normaliser le gradient. Cette normalisation équilibre la taille du pas (momentum), en diminuant le pas pour les grands gradients pour éviter d'exploser et en augmentant le pas pour les petits gradients pour éviter de disparaître.

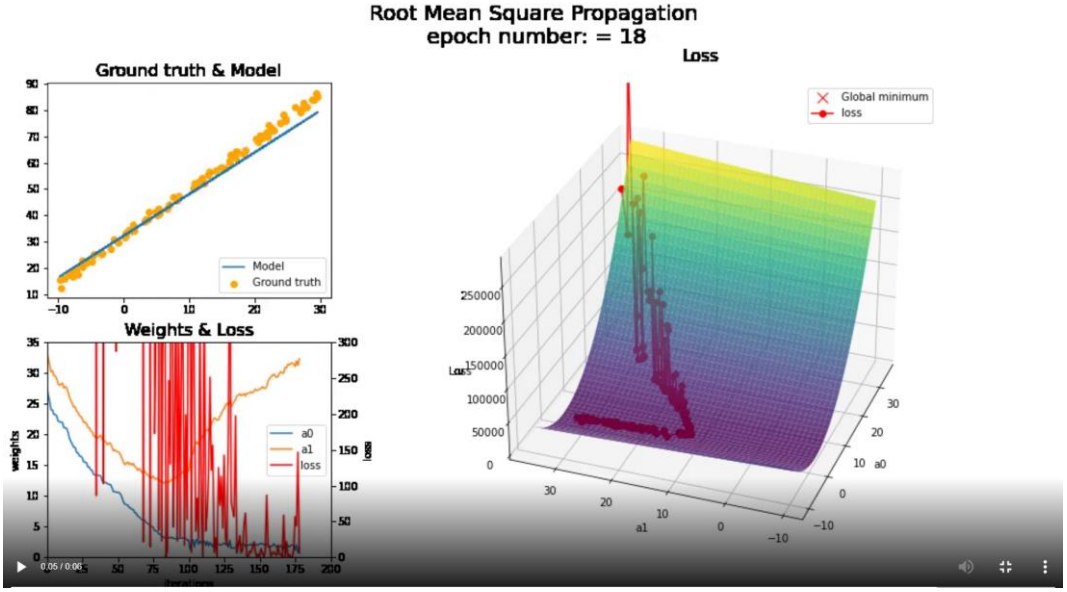

Figure 5 : Animations montrant la performance de l'algorithme RMSProp

Paramètres (alpha/epochs/rho)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
0.5/200/0.9	0.054901	193	Converge lentement et instable à la convergence.
0.1/300/0.9	0.085771	Ne converge pas	Converge trop lentement
0.5/300/0.6	0.107719	252	Converge trop lentement et encore

			plus instable à la convergence
0.5/200/0.7	0.062831	221	Converge lentement et instable à la convergence

I.d.v. - SGD avec Momentum

Particularité : l'idée est de calculer à la volée une moyenne mobile des gradients précédents, et d'utiliser cette moyenne mobile, et non le gradient de l'exemple courant, dans la formule de mise à jour.

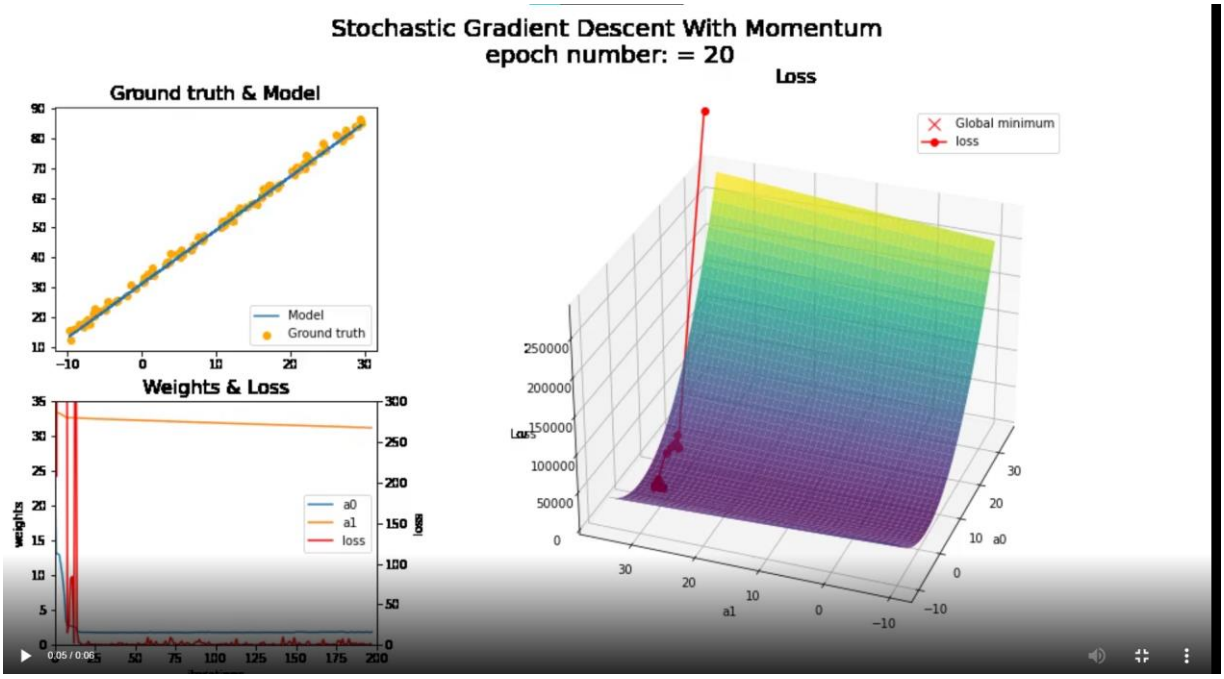

Figure 6 : Animations montrant la performance de l'algorithme SGDM

Paramètres (alpha/epochs/mu)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
0.001/200/0.24	0.042885	60	Converge rapidement et stable à la convergence.
0.005/200/0.24	0.050898	24	Converge rapidement mais ne reste pas à la convergence à cause des sauts

0.001/200/0.5	0.056849	31	Converge rapidement mais présence de saut de valeurs
0.003/200/0.5	0.064828	35	Converge rapidement mais présence de saut de valeurs

I.d.vi. - ADAM

Particularité : Combinaison de AdaGrad et RMSProp.

Au lieu d'adapter les taux d'apprentissage des paramètres en fonction du premier moment moyen (la moyenne) comme dans RMSProp, Adam utilise également la moyenne des seconds moments des gradients (la variance non centrée).

Plus précisément, l'algorithme calcule une moyenne mobile exponentielle du gradient et du gradient au carré, et les paramètres beta1 et beta2 contrôlent les taux de décroissance de ces moyennes mobiles.

La valeur initiale des moyennes mobiles et les valeurs bêta1 et bêta2 proches de 1,0 (recommandé) entraînent un biais des estimations de moment vers zéro. Ce biais est surmonté en calculant d'abord les estimations biaisées avant de calculer ensuite les estimations corrigées du biais.

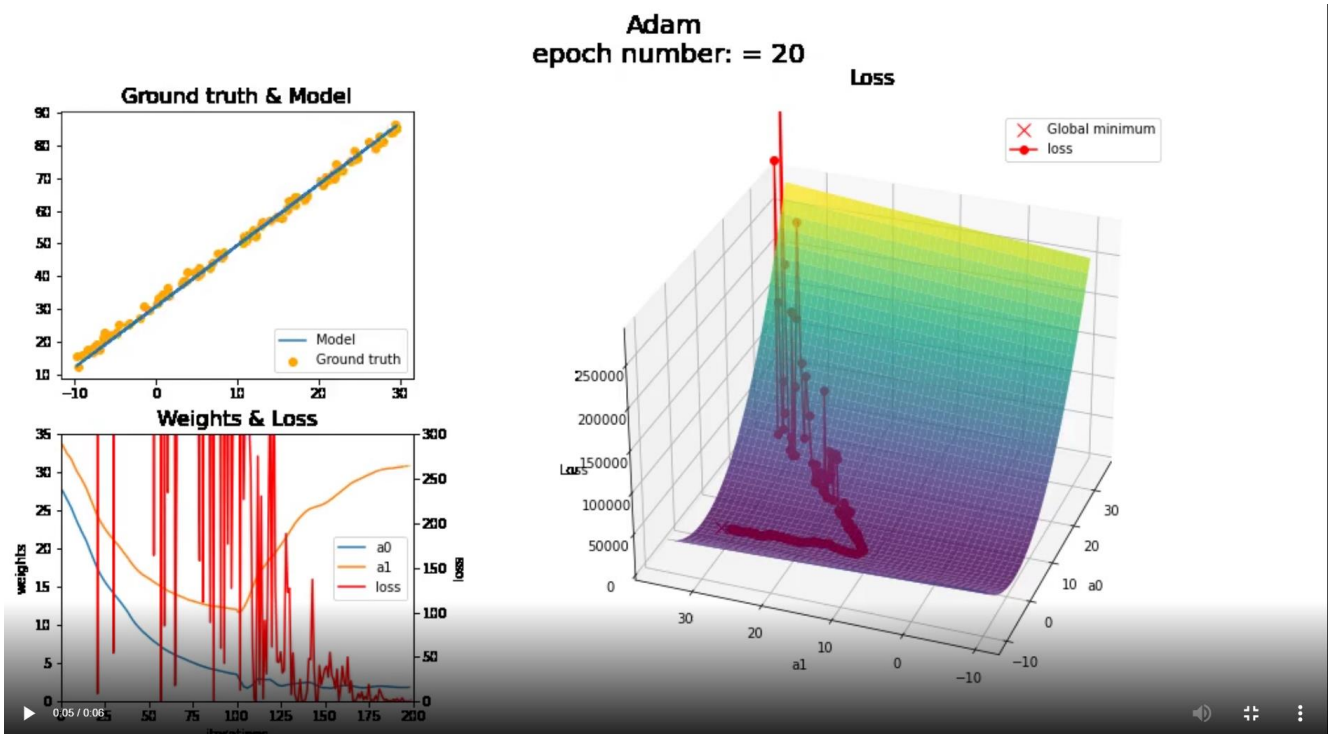

Figure 7 : Animations montrant la performance de l'algorithme Adam

Paramètres (alpha/epochs/b1/b2)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution
0.38/200/0.9/0.999	0.064796	198	Converge lentement et stable à la convergence.
0.38/200/0.5/0.999	0.059818	Pas de convergence	Trop lente
0.38/1000/0.5/0.999	0.338096	620	Converge trop lentement mais très stable
4/500/0.5/0.999	0.138635	47	Converge rapidement mais présence de saut de valeurs

I.e. - Conclusion sur l'influence du choix de l'algorithme et de leurs paramètres

Algorithmes (aux meilleurs paramètres trouvés)	Temps de calcul(secondes)	Nombre d'itérations nécessaires à la convergence	Qualité de la solution	Sensibilité des paramètres
BGD	0.00398	7	Converge très rapidement et stable	Sensible (alpha*2=divergence)
SGD	0.032911	46	Converge rapidement et stable	Sensible (10*alpha=convergence mais relation quand même instable)
AdaGrad	0.058863	197	Converge lentement et stablement	Ne change pas la convergence mais le temps de convergence donc peu sensible
RMSProp	0.054901	193	Converge lentement et instable à la convergence.	Rho et alpha peu sensible

SGDM	0.042885	60	Converge rapidement et stable à la convergence.	Alpha sensible, mu peu sensible
ADAM	0.064796	198	Converge lentement et stable à la convergence.	Alpha sensible, b1 et b2 sensible dans le temps de convergence mais pas la stabilité.

Comme on a pu le voir, ce sont les algorithmes de descente du gradient les moins complexes qui marchent au mieux pour prédire la relation degrés Celsius Fahrenheit. La raison principale à cela c'est que les algorithmes complexes comme ADAM ou RMSProp sont moins stables que les algos simples comme SGD ou BGD. Les paramètres des algos complexes sont plus sensibles et prennent du temps à être bien paramétrer pour avoir un résultat aussi efficace que dans les derniers algos cités.

Pour une régression linéaire simple comme la nôtre (avec 2 paramètres), BGD et SGD sont extrêmement efficaces. La taille de l'échantillon est bien choisie pour SGD.

Ce qu'il faut retenir :

- Globalement, bien choisir le taux d'apprentissage est extrêmement important pour la stabilité et la rapidité de la convergence de l'algorithme. Un alpha grand convergera rapidement mais sera instable au point de la solution car les gradients changeront trop. Un alpha démesurément grand fera que le minimum global de la fonction de cout est évité et que les paramètres divergeront vers l'infini. Un alpha trop petit mettra très longtemps à converger. Il faut donc trouver le bon alpha qui converge rapidement mais pas trop grand pour bien trouver le minimum global de la fonction de cout.
- Le Batch Size pour SGD est important car ce qui va déterminer la qualité de cet algorithme. Si l'échantillon aléatoirement choisi est trop petit, on va faire du surapprentissage pour cet échantillon en particulier. S'il est trop grand, l'apprentissage donnera qu'une formule globale des données sans être trop précis.
- Le momentum choisi pour SGDM peut faire diverger les paramètres s'il est trop grand. Le momentum est la vitesse initiale donné au système donc il a les mêmes effets qu'alpha quant à la convergence ou à la divergence.
- RMSProp prend en compte les anciens gradients pour calculer les nouveaux, le paramètre rho décide de la proportion de l'ancien gradient à garde donc la aussi, ce paramètre ne doit pas être trop petit sinon l'algo n'a plus d'interet mais ni trop grand sinon le temps de convergence sera trop long
- Même conclusion pour Adam.

I.f. - Recherche d'or dans la cordillère des Andes

Le bureau de recherches géologique et minière en charge de la gestion durable des ressources naturelles souhaite mieux comprendre la formation de l'or et mieux prédire la présence ou l'absence de ce minerai. Pour cela, il a constitué à partir d'un système d'information géographique une banque de données constituée de 1000 individus d´écrits par 24 variables, dont 21 sont quantitatives (20

variables géophysiques et le numéro du site) et 3 sont qualitatives (l'âge de la roche à 3 modalités, le type de la roche à 2 modalités et le type de site à 2 modalités). On cherche en particulier à prédire efficacement la valeur de la variable qualitative site qui informe de la présence ("gisement") ou de l'absence ("stérile") du minerai pour chacun des 1000 sites connus. Le but de la modélisation est de concevoir et d'entraîner un (ou plusieurs) modèles prédictifs pour prédire au mieux la présence ou l'absence d'or. Nous avons deux banques de données :

- gisementLearn.txt : contient des données d'apprentissage (étiquetées).
- gisementTestNolabel.txt : contient des données à prédire (non étiquetées).

I.f.vii. - Statistique descriptive des données

DIST_180	DIST_22	DIST_45	DIST_67	DIST_90	AGE	ROCK	OR
50680.140	7843.3200	859.0046	45221.5900	159401.4000	PALEOZOIC	VOLCANIC	STERILE
86875.320	9609.7860	18296.4600	6855.5566	361148.9400	PROTEROZOIC	VOL_SEDIMENTARY	STERILE
8380.743	119434.7700	125015.3050	28738.4180	443106.7500	PALEOZOIC	METAMORPHIC	STERILE
12625.100	28468.9790	45750.2660	6823.7417	59060.4450	PROTEROZOIC	PLUTONIC	STERILE
20383.918	5360.7373	31686.1560	129477.4000	112078.0900	PROTEROZOIC	PLUTONIC	STERILE
...	...	...	...	...	...	...	...
17734.870	152081.7000	320008.9400	153881.9400	1280.1881	PALEOZOIC	SEDIMENTARY	STERILE
22874.299	41015.2930	196306.4200	10720.1640	31846.4820	PROTEROZOIC	VOLCANIC	STERILE
74368.860	101930.6700	109869.7600	47960.8670	56696.8630	PROTEROZOIC	VOLCANIC	STERILE
103769.195	159271.9500	381233.7800	54258.8360	195570.4500	PALEOZOIC	SEDIMENTARY	STERILE
103551.250	159030.0500	381082.5300	53884.3870	195327.6000	PALEOZOIC	SEDIMENTARY	STERILE

Figure 8: Données du fichier gisementLearn.txt

On a 1633 individus décrits par 20 variables quantitatives et 3 qualitatives. La première chose que l'on va faire est d'encodé les variables labélisés avec un *LabelEncoder().*

```
label_encoder = preprocessing.LabelEncoder()
data["AGE"]=label_encoder.fit_transform(data["AGE"])
data["ROCK"]=label_encoder.fit_transform(data["ROCK"])
data["OR"]=label_encoder.fit_transform(data["OR"])
```

Figure 9: Implémentation du label Encoder

Le label encoder va attribuer une valeur numérique à chaque type de label dans chaque variable.

On peut analyser nos données en plotant un histogramme de chaque variable :

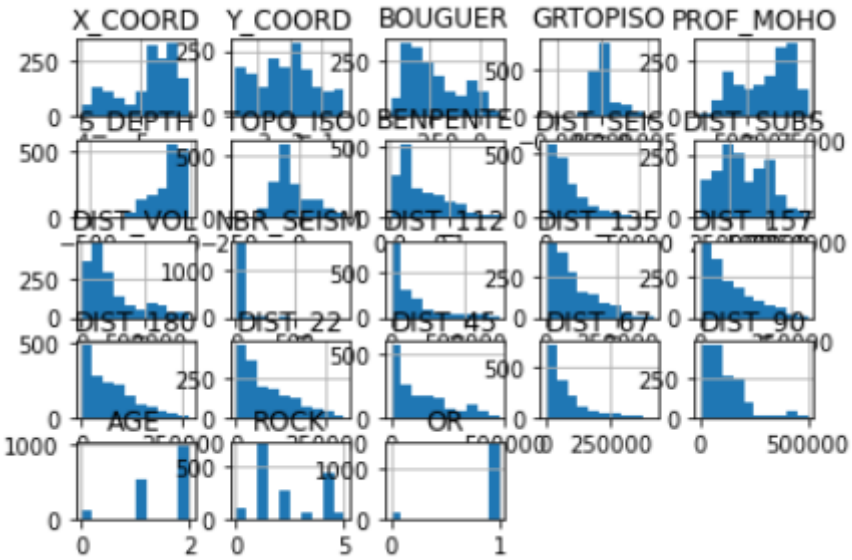

Figure 10: Histogrammes des variables dans les données

On remarque directement quelque chose : les données ne sont pas du tout équilibrées. Aucun histogramme est équilibré. Surtout pour la variable OR que l'on cherche à prédire qui a à peine 10% de 0 (ce qui signifie présence d'or) ce qui posera un problème au moment de l'apprentissage puisque le modèle risque d'apprendre très bien à prédire qu'il n'y pas d'or mais moins bien pour prédire qu'il y en a.

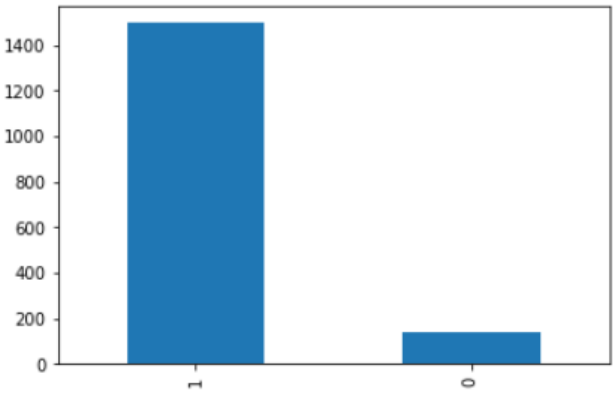

Figure 11: Représentants de la variable OR

En affichant les boites à moustaches de chaque variable (ici la variable DIST_112) :

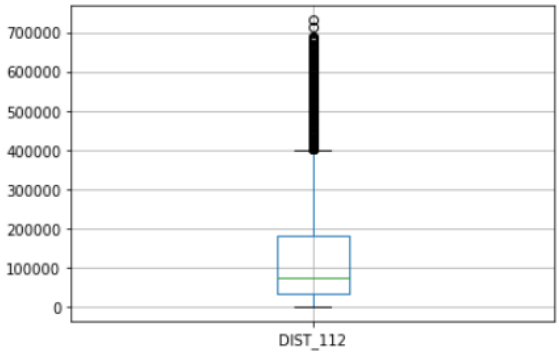

Figure 12: Boite à moustaches de la variable DIST_112

On remarque que la plupart des variables ont des valeurs aberrantes dans le dataset (ici les cercles noir). Cela peut être une raison également que l'apprentissage ne marche pas bien. On y reviendra.

On va maintenant séparer nos données en 4 variables X_train, X_test, y_train et y_test. Les datasets y_train et y_test représentent le paramètre que l'on cherche à prédire c'est-à-dire le paramètre « OR » et X_train, X_test seront le reste des données qui serviront à l'apprentissage supervisée. Les datasets *train* serviront à entrainer le modèle et les datasets *test* serviront à tester l'efficacité et la réplicabilité du modèle appris.

```
X=data.drop(columns=['OR'])
y=data['OR']


X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.33, shuffle=True)
```

Figure 13: Séparation en jeux de données d'apprentissage et de validation

On choisit que la taille des données de test sont de 33%* la taille des données totales. Ce qui nous permettra d'apprendre sur 2/3 des données et de valider sur 1/3. L'attribut *shuffle=true* permet de dire de mélanger le dataset avant de séparer en échantillons.

En examinant les données, on se rend compte qu'il y a une énorme différence d'ordre de grandeurs entre les données elles-mêmes, il suffit de regarder la variance des données :

```
data.var()
```

```
X_COORD      2.251605e+11
Y_COORD      8.009564e+11
BOUGUER       1.667143e+04
GRTOPISO      1.599577e-06
PROF_MOHO     1.025979e+08
S_DEPTH       5.356953e+03
TOPO_ISO      4.070134e+03
BENPENTE      1.457432e-03
DIST_SEIS     8.936634e+07
DIST_SUBS     2.737663e+10
DIST_VOL      3.891478e+10
NBR_SEISM     5.120194e+03
DIST_112      2.506202e+10
DIST_135      6.014005e+09
DIST_157      4.269900e+09
DIST_180      3.587042e+09
DIST_22       4.793832e+09
DIST_45       1.391661e+10
DIST_67       6.535482e+09
DIST_90       8.625245e+09
AGE           3.895716e-01
ROCK          2.101816e+00
OR            7.690345e-02
```

Figure 14: Variance de chaque variable du jeu de données

Pour palier à ce problème, on va normaliser les données avec un *StandartScaler()*.

```
scaler = StandardScaler()
# we fit the train data
scaler.fit(X_train)
# scaling the train data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 15: Implémentation de la normalisation des données

Après application :

```
print(X_train.var())
print(X_test.var())
```

```
1.0
1.0000559217460911
```

Figure 16: Vérification de la normalisation des données

La variance est bien de 1 donc les données sont bet et bien normalisées maintenant.

Apprentissage du modèle grâce au perceptron multi-couches :

Pour apprendre notre modèle nous allons utiliser un perceptron multi-couches : MLPClassifier(). Le perceptron demande un certain nombre de paramètres à utiliser. Pour trouver les meilleurs paramètres pour notre modèle et ainsi obtenir les meilleurs scores de prédiction, nous allons utiliser la fonctionnalité *GridSearchCV()* de scikit-learn qui permet de tester plusieurs paramètres en une simulation et de nous dire quels paramètres nous permet d'obtenir les meilleurs résultats :

```
parameters = {'solver':('adam', 'sgd'), 'alpha':[0.0001, 0.005,0.38], 'learning_rate':('constant','adaptive'),'max_iter':[1000]}

#MLPClassifier(hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='co

MLP=MLPClassifier()
clf = GridSearchCV(MLP, parameters)
clf.fit(X_train,y_train)
```

Figure 17: Implémentation du gridSearch

Nous demandons donc au MLP de tester l'algorithme *Adam et SGD* pour des valeurs de alpha différente et un pas d'apprentissage constant et adaptatif. Avec 1000 itérations maximales.

D'après notre GridSearch, voici les meilleurs paramètres :

```
clf.best_params_
```

```
{'alpha': 0.005,
 'learning_rate': 'adaptive',
 'max_iter': 1000,
 'solver': 'sgd'}
```

Figure 18: Meilleurs paramètres sortis par le gridSearch

Ces paramètres sont plutôt cohérents et sont en accord avec les résultats obtenus dans la partie 1. Si ce n'est que le pas d'apprentissage ne peut être adaptatif dans le SGD. Je pense que l'algorithme utilise automatiquement lr='constant'

Et voici la fonction de perte avec ces paramètres :

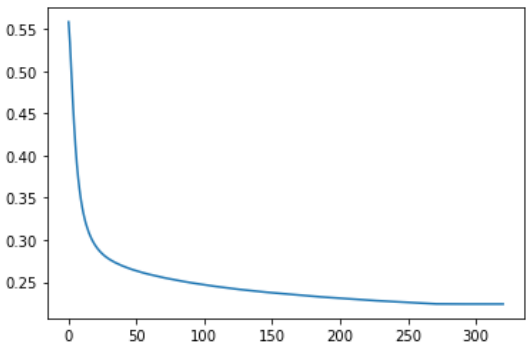

Figure 19: Fonction de perte pour le modèle décidé par le gridSearch

Rien d'anormal jusqu'ici.

Mais en testant ces paramètres sur nos datasets de validation

```
plt.figure()
plt.plot(clf.loss_curve_)
y_test_pred = clf.predict(X_test)
#clf.score(X_test, y_test)
Y_train_pred=clf.predict(X_train)
print(accuracy_score(Y_train_pred, y_train))
print(accuracy_score(y_test_pred, y_test))
```

Figure 20: implémentation du test de validation du modèle

On obtient la matrice de confusion et le rapport de prédiction :

```
[[  2   0]
 [ 50 487]]
```

	precision	recall	f1-score	support
0	0.04	1.00	0.07	2
1	1.00	0.91	0.95	537
accuracy			0.91	539
macro avg	0.52	0.95	0.51	539
weighted avg	1.00	0.91	0.95	539

Figure 21: Matrice de confusion et rapport de prédiction du modèle du gridSearch

On voit tout de suite dans le rapport de prédiction qu'on a uniquement 2 cas ou il y a de l'or dans les données de validation. Ceci est dû à la taille des sous-échantillons choisies par le SGD on privilégiera alors l'algo ADAM pour pouvoir décider de la qualité de notre modèle. De plus il n'y a pas grande différence lorsqu'on regarde les scores obtenus par les différents algos de notre GridSearch.

Lorsqu'on passe à ce modèle :

```
clf=MLPClassifier(alpha=0.01,learning_rate='adaptive',max_iter=1000,solver='adam')
```

On obtient la matrice et le rapport suivant :

```
[[ 17  16]
 [ 35 471]]
              precision    recall  f1-score   support

           0       0.33      0.52      0.40        33
           1       0.97      0.93      0.95       506

    accuracy                           0.91       539
   macro avg       0.65      0.72      0.67       539
weighted avg       0.93      0.91      0.92       539
```

Figure 22: Matrice de confusion et rapport de prédiction du modèle du adam

On conclu qu'on a une précision et un rappel trop faible quand il y a une présence d'or dans le gisement.

Pour essayer d'améliorer notre résultat on va utiliser l'oversampling (sur-échantillonage).

Utilisation de l'OverSampling :

l'oversampling va permettre de rééquilibrer le dataset. La technique utilisée est de générer des points de données étroitement liées avec les données déséquilibrer en question. Par exemple, pour l'or, l'algorithme d'oversampling va générer des points de données ou il y aura de l'or dans la colonne « OR » tout en faisant en sorte que les points générés ressemblent réellement aux points ou il y a de l'or dans le dataset.

Il existe plusieurs techniques d'oversampling, on va toutes les tester dans le code mais d'après la documentation que j'ai pu lire voici celle qui me semblerait la plus cohérente :

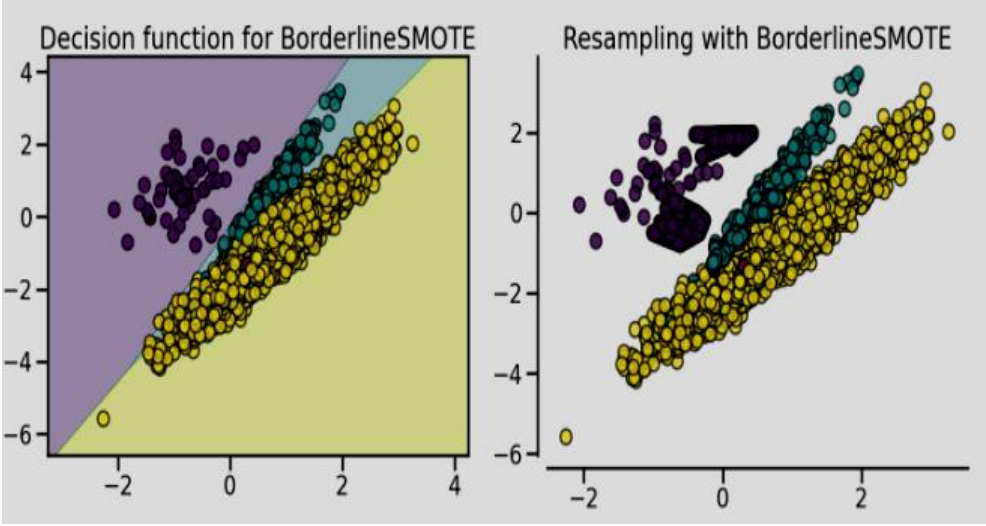

Figure 23: BorderlineSMOTE oversampling

Ainsi, on pourrait générer des données proches de celles et uniquement de celles ou il y a de l'or.

En utilisant l'over sampling :

```
from imblearn.over_sampling import BorderlineSMOTE
X_resampled, y_resampled = BorderlineSMOTE().fit_resample(X_train, y_train)
clf.fit(X_resampled,y_resampled)
plt.figure()
plt.plot(clf.loss_curve_)
y_test_pred = clf.predict(X_test)
print(accuracy_score(y_test_pred, y_test))
print(classification_report(y_test_pred, y_test))
```

Figure 24: Implémentation de l'oversampling

J'obtiens :

```
0.8998144712430427
              precision    recall  f1-score   support

           0       0.44      0.48      0.46        48
           1       0.95      0.94      0.94       491

    accuracy                           0.90       539
   macro avg       0.70      0.71      0.70       539
weighted avg       0.90      0.90      0.90       539
```

Figure 25: Rapport de prédiction du modèle adam après l'oversampling

On remarque qu'il y a plus de support pour la présence d'or dans le gisement et par conséquent on arrive à des résultats meilleurs que précédemment sans over sampling. On peut voir que les données ont été rééquilibrés :

```
y_resampled.hist()
```

```
<AxesSubplot:>
```

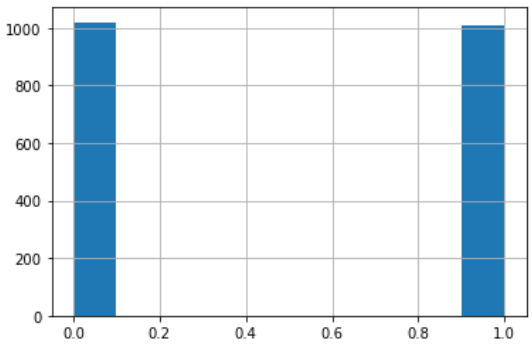

Figure 26: Nombre de représentants de la variable OR après oversampling

Il existe aussi des techniques d'undersampling et une combinaison d'under et d'over sampling mais selon moi ces techniques n'a pas lieu d'être dans notre cas car l'undersampling notamment réduirait le nombre de données représenter et on aurait encore moins de données pour apprendre.

Conclusion :

Les résultats finaux obtenus sont :

0.8998144712430427

	precision	recall	f1-score	support
0	0.44	0.48	0.46	48
1	0.95	0.94	0.94	491
accuracy			0.90	539
macro avg	0.70	0.71	0.70	539
weighted avg	0.90	0.90	0.90	539

```
[[ 17  29]
 [ 25 468]]
```

Figure 25: Rapport de prédiction du modèle adam après l'oversampling

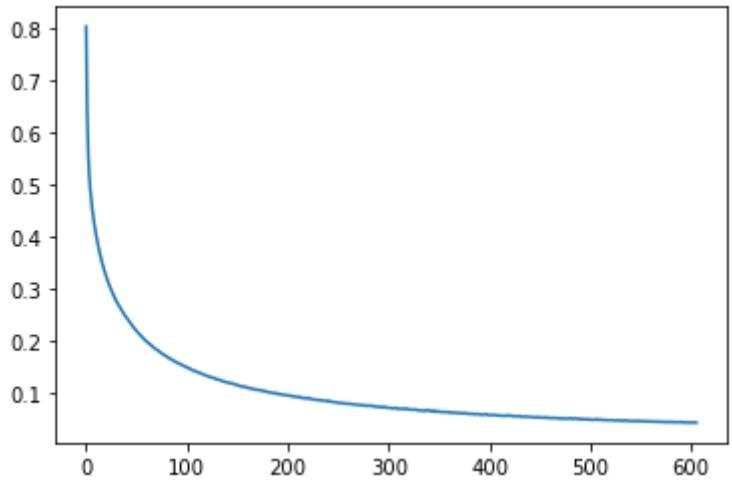

Figure 25: Fonction de perte du modèle adam après l'oversampling

Premièrement, la fonction de perte a une allure parfaite : elle converge rapidement vers 0 et ne fait pas de sur-apprentissage.

Secondement, la matrice de confusion est bonne pour prédire qu'il n'y a pas d'or (on a 25 de faux négatif pour 468 vrai négatif) mais beaucoup moins bonne pour prédire qu'il y a de l'or (17 vrai négatif contre 29 faux négatifs).

Pour rappel, la précision est le nombre de vrais positifs / le nombre de vrais positifs + faux positifs

Et le rappel est le nombre de vrais positifs / le nombre de vrais positifs + le nombre de faux négatifs.

Ainsi, il est important que ces deux valeurs soient hautes en même temps car elles nous disent autant toutes les deux sur la fiabilité du modèle.

Pourquoi n'arrive-t-on pas à avoir un meilleur rappel et une meilleure précision pour prédire quand il y a de l'or dans le gisement ?

Je pense que le fait que les données ou on a de l'or soit sous-représenté empêche l'algorithme de bien déterminer s'il y a de l'or ou pas dans le gisement. C'est d'ailleurs logique puisque moins il y a de données d'apprentissage pour une classe en particulier, moins cette classe sera distincte pendant la validation.

On a essayé de corriger ce problème avec l'oversampling, cependant les variables étant nombreuses et précises, les données générées permettent effectivement d'apprendre mieux mais le résultat n'est toujours pas satisfaisant (50% de bons résultats) car les données générées peuvent ressembler trop à un cas où il n'y a pas d'or.

Pour augmenter encore la précision et le rappel de notre modèle on pourrait faire 2 choses :

- Enlever toutes les valeurs aberrantes montrées précédemment. Mais cette méthode prendrait du temps et nous enlèverait encore des données (ce qu'on souhaite éviter).
- Ajouter de nouvelles données de présence d'or dans les gisements dans le datasets