CHAPITRE 1

QUELQUES RAPPELS THÉORIQUES SUR L'AUTOENCODER

1.1 Définition de l'AutoEncoder

L'autoencoder à l'instar de l'ACP et la FAMD est une méthode de réduction de dimension qui se rapproche beaucoup plus de la FAMD en ce sens qu'elle s'applique à des données mixtes c'est à dire un ensemble de variables qualitatives et quantitatives. Plus précisement, un autoencodeur (autoencoder) est un type de réseau de neurones artificiel utilisé pour l'apprentissage non supervisé, dont l'objectif principal est de compresser et reconstruire des données d'entrée (ensembles de données quantitatives et qualitatives).

1.2 Différentes parties d'un autoencodeur

On peut décomposer un auto-encodeur en trois parties :

— l'**Encodeur** : L'encodeur transforme l'entrée en une représentation dans un espace de dimension plus faible appelé espace latent. L'encodeur compresse donc l'entrée dans une représentation moins coûteuse.

— le **Goulot d'étranglement (ou bottleneck)** : Cette partie du réseau contient la représentation compressée de l'entrée qui est sera introduite dans le décodeur.

— le **Décodeur** : cette partie doit construire l'output à l'aide de la représentation latente de l'entrée.

:

1.3 Hyperparamètre d'un autoencodeur

Un hyperparamètre d'un modèle de machine learning est un paramètre dont la valeur est fixée avant l'entraînement du modèle et qui ne se met pas à jour lors de l'apprentissage. Contrairement aux paramètres

du modèle (comme les poids d'un réseau de neurones ou les coefficients d'une régression), qui sont ajustés par le processus d'entraînement, les hyperparamètres contrôlent le comportement global du modèle et son processus d'optimisation.

Pour un autoencodeur classique, les principaux hyperparamètres sont liés à l'architecture, l'optimisation et la régularisation. Voici une liste détaillée :

— **La normalisation** Le choix de la normalisation est un hyperparamètre crucial pour l'entraînement d'un autoencodeur, influençant la stabilité et la performance du modèle. Les deux méthodes les plus couramment utilisées sont le Min-Max Scaling et la Standardisation (Z-score). Le Min-Max Scaling transforme les données pour les ramener dans une plage $[0, 1]$, ce qui est particulièrement adapté aux modèles utilisant une activation sigmoïde en sortie. En revanche, la Standardisation centre les données autour de 0 avec un écart-type de 1, facilitant la convergence des réseaux neuronaux, notamment avec des activations comme ReLU ou Tanh. Le choix entre ces deux méthodes dépend de la distribution des données et doit être testé empiriquement pour obtenir les meilleures performances.

— **Nombre de couches cachées (dans l'encodeur et le décodeur)** : Plus il y a de couches, plus le modèle peut apprendre des représentations complexes, mais un excès peut rendre l'entraînement instable.

— **Nombre de neurones par couche** : En général, on réduit progressivement le nombre de neurones dans l'encodeur jusqu'à la couche latente, puis on les augmente dans le décodeur pour reconstruire l'entrée.

— **Taille de la couche latente (bottleneck)** : C'est un hyperparamètre clé qui contrôle le degré de compression de l'information. Une valeur trop petite peut entraîner une perte d'information, tandis qu'une valeur trop grande peut empêcher l'autoencodeur d'apprendre une bonne représentation.

— **Fonction d'activation** : ReLU, LeakyReLU, Sigmoid, Tanh... ReLU est souvent un bon choix, sauf pour la couche de sortie où on utilise souvent Sigmoid ou Tanh selon la normalisation des données.

— **Fonction de perte** : lors de l'entraînement d'un auto-encodeur, la fonction de perte, qui mesure la perte de reconstruction entre la sortie et l'entrée, est utilisée pour optimiser les poids du modèle via la descente de gradient lors de la rétropropagation. Le ou les algorithmes idéaux pour la fonction de perte dépendent de la tâche pour laquelle l'auto-encodeur sera utilisé.

— **Régularisation pour éviter le surajustement**

  ✓ L1/L2 Regularization (Weight Decay) → Ajoute une pénalité sur les poids pour éviter un apprentissage trop spécifique aux données d'entraînement.

- ✓ Dropout → Fait en sorte que certains neurones ne soient pas activés lors de l'entraînement, ce qui aide à la généralisation.
- ✓ Batch Normalization → Accélère la convergence et stabilise l'entraînement.

1.4 Retour sur les fonctions d'activations

1.4.1 Définition

La forward propagation est le processus de propagation et de transformation des données à travers un réseau de neurones. Ce processus permet à un réseau de neurones de produire un résultat à partir de données. Avant la production du résultat, les données passent à travers le modèle. Chaque neurone reçoit alors les données et leur applique une transformation. Cette succession de transformations permet de produire un résultat. La transformation évoquée est une opération mathématique réalisée par un neurone grâce à sa fonction d'activation et son poids.

Plus simplement la fonction d'activation est l'attribut du neurone lui permettant de transformer une donnée. Elle joue un rôle central dans un réseau de neurones car elle permet d'appliquer une transformation non linéaire.

Une transformation non-linéaire modifie la représentation spatiale des données. Cette transformation nous donne la capacité d'explorer les données sous différentes perspectives et ainsi, de mieux les comprendre. À l'inverse, une transformation linéaire permet, certes, de modifier les données, mais n'a pas d'influence sur leur représentation spatiale.

Pour illustrer cela, imaginons que nous ayons deux valeurs : 13 et 54. Ces valeurs ont une représentation spatiale : 13 est inférieur à 54. Si nous appliquons une opération linéaire, par exemple une multiplication par 2, la valeur de ces données sera alors modifiée. Néanmoins, leur position spatiale restera inchangée : 13x2 = 26 sera toujours inférieur à 54x2 = 108. Les transformations linéaires n'ont donc pas d'impact sur la représentation spatiale des données. Mais les transformations non linéaires en ont un.

Si nous appliquons une opération non linéaire à 13 et 54, par exemple la fonction cosinus, cela modifiera la valeur des données, mais également leur position spatiale : cos(13) ≈ 0.90 est supérieur à cos(54) ≈ -0.82.

Cette propriété des fonctions non linéaires est un atout majeur, c'est pourquoi les réseaux de neurones l'exploitent par l'intermédiaire des fonctions d'activation.

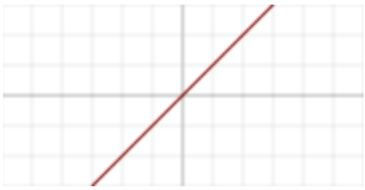
1.4.2 Exemples usuels de fontions d'activations

En mathématiques, une quantité indénombrable de fonctions non linéaires existe. Néanmoins, seulement une petite fraction d'entre elles est utilisée comme fonction d'activation en Deep Learning.

La différence entre ces fonctions d'activation peut parfois sembler négligeable, cependant, certaines de ces fonctions doivent être choisies avec précision si l'on souhaite créer un réseau de neurones fonctionnel.

Linear

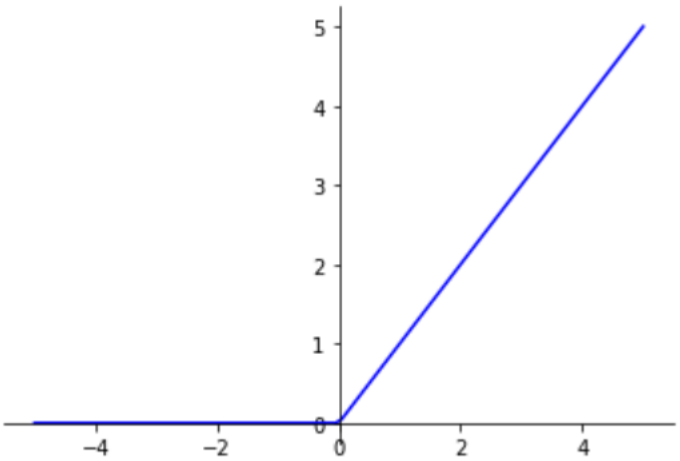
Utilisé en couche de sortie pour une utilisation pour une régression. On peut la caractériser de nulle, puisque les unités de sortie seront identiques à leur niveau d'entré. Intervalle de sortie $(-\infty; +\infty)$. Elle équivaut en faites à ne pas utiliser de fonction d'activation.



Relu

La fonction Rectified Linear Unit (ReLU) est la fonction d'activation la plus couramment utilisée en Deep Learning. Elle donne x si x est supérieur à 0, 0 sinon. Autrement dit, c'est le maximum entre x et 0 :

fonction_ReLU(x) = max(x, 0)



Fonction ReLU – Rectified Linear Unit

Cette fonction permet d'appliquer un filtre en sortie de couche. Elle laisse passer les valeurs positives dans les couches suivantes et bloque les valeurs négatives. Ce filtre permet alors au modèle de se concentrer uniquement sur certaines caractéristiques des données, les autres étant éliminées.

Fonction Leaky ReLU

La fonction Leaky ReLU est une variante de ReLU qui vise à résoudre le problème des « neurones morts ». Sa formule est :

$$f(x) = \begin{cases} x, & \text{si } x \geq 0 \\ \alpha x, & \text{si } x < 0 \end{cases}$$

où α est une petite constante, souvent égale à 0.01.

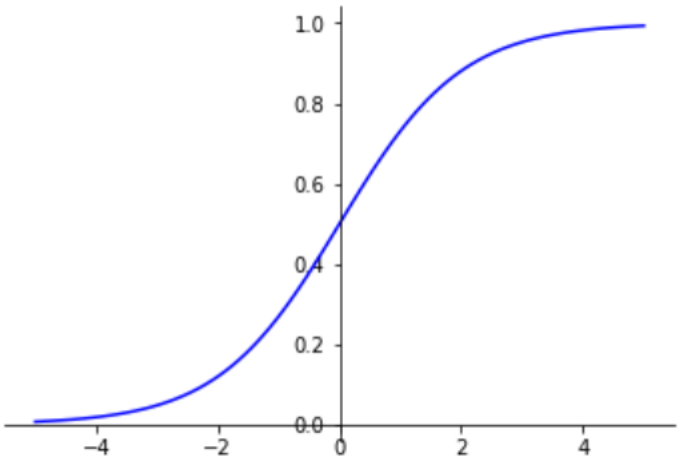
Cette faible pente pour les valeurs négatives permet aux neurones de continuer à apprendre même lorsque les entrées sont négatives, évitant ainsi la mort des neurones.

Sigmoid

La fonction Sigmoïde est la fonction d'activation utilisée en dernière couche d'un réseau de neurones construit pour effectuer une tâche de classification binaire.

Elle donne une valeur entre 0 et 1.

$$\text{fonction\_Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$



Fonction Sigmoïde

Cette valeur peut être interprétée comme une probabilité. Dans une classification binaire, la fonction d'activation sigmoïde permet alors d'obtenir, pour une donnée, la probabilité d'appartenir à une classe.

Plus le résultat de la sigmoïde est proche de 1, plus le modèle considère que la critique est positive. Inversement, plus le résultat de la sigmoïde est proche de 0, plus le modèle considère que la critique est négative.

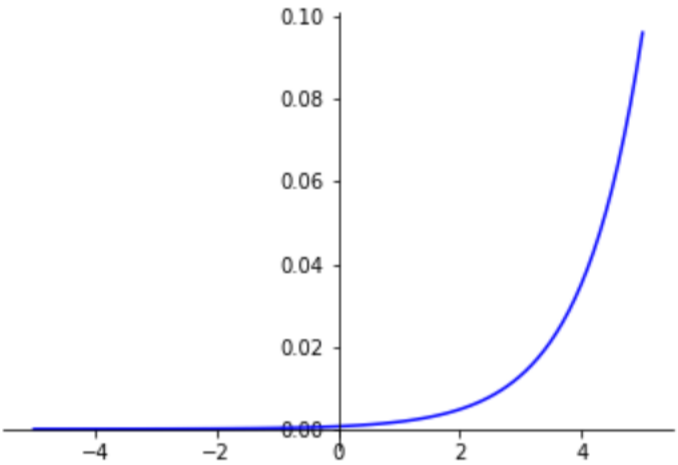
La fonction d'activation sigmoïde permet donc d'obtenir un résultat ambivalent, donnant une indication sur deux classes à la fois.

Softmax

La fonction Softmax est la fonction d'activation utilisée en dernière couche d'un réseau de neurones construit pour effectuer une tâche de classification multi-classes.

Pour chaque sortie, Softmax donne un résultat entre 0 et 1. De plus, si l'on additionne ces sorties entre elles, le résultat donne 1.

$$\text{fonction\_Softmax}(x) = \frac{e^x}{\sum e^{x_i}}$$



Fonction Softmax

Comme pour la fonction Sigmoïde, les valeurs résultantes de l'utilisation de la fonction Softmax peuvent être interprétées comme des probabilités. Dans ce cas, chacune de ces valeurs est associée à une classe du dataset.

Il est important de noter que nous n'aurions pas pu utiliser la fonction Sigmoïde en dernière couche d'un réseau de neurones pour une tâche de classification multi-classes. Elle donnerait bien, pour chaque élément, une valeur entre 0 et 1, mais ces éléments additionnés pourraient ne pas être égaux à 1. Ainsi, le résultat ne respecterait pas les lois de probabilité et serait donc fallacieux.

En revanche, la fonction Softmax, grâce à sa caractéristique de produire des résultats qui, additionnés, donnent 1, respecte les lois de probabilité. Elle est donc le noyau dur d'un réseau de neurones construit pour effectuer une tâche de classification multi-classes.

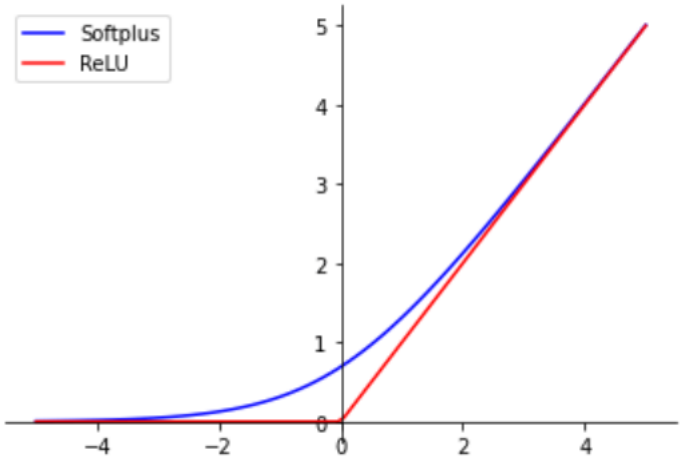
Softplus

La fonction Softplus est une approximation « lisse » (en anglais « soft ») de la fonction ReLU.

Lorsque les valeurs d'entrée sont positives, la fonction Softplus se comporte, à quelques exceptions près, de la même manière que ReLU. Toutefois, pour des valeurs d'entrée négatives, la fonction Softplus

n'applique pas de filtre comme ReLU, mais les fait tendre vers zéro.

$$\text{fonction\_Softplus}(x) = \log(e^x + 1)$$



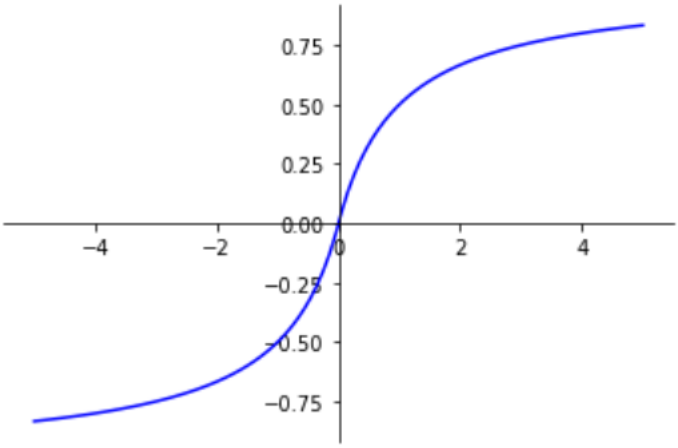
Fonction Softplus

Softsign

La fonction Softsign permet d'appliquer une normalisation aux valeurs d'entrée.

Elle permet d'obtenir des valeurs de sortie entre -1 et 1

$$\text{Softsign}(x) = \frac{x}{|x| + 1}$$



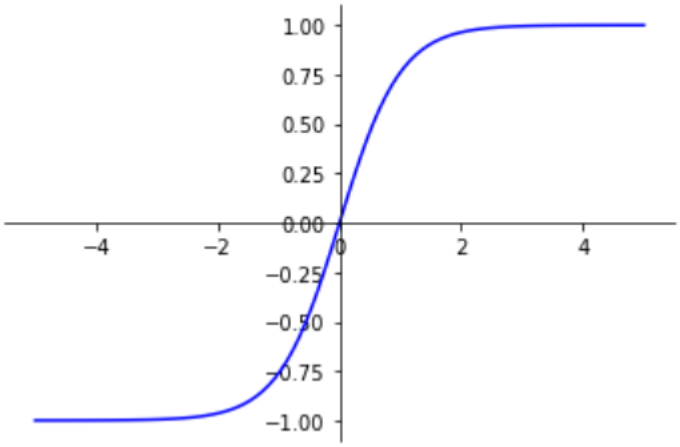
Fonction Softsign

La fonction Softsign permet également de préserver le signe (positif ou négatif) des valeurs d'entrée. Une propriété qui peut s'avérer utile dans certaines tâches, mais dont l'utilisation reste peu commune.

tanh

La fonction tanh permet d'appliquer une normalisation aux valeurs d'entrée. Elle peut également être utilisée au lieu de la fonction Sigmoïde dans la dernière couche d'un modèle de classification binaire Elle donne un résultat entre -1 et 1.

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$



Fonction tanh

Lors d'une classification binaire, on ne peut pas directement interpréter le résultat obtenu après utilisation de la fonction tanh comme une probabilité car le résultat se situera entre -1 et 1. On pourra alors le modifier pour l'observer de la même manière qu'une probabilité.

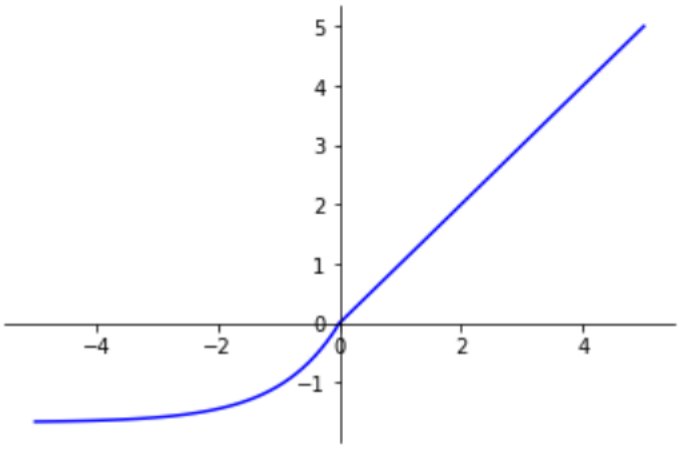
ELU

La fonction Exponential Linear Unit (ELU) est une déclinaison de la fonction ReLU.

ELU se comporte de la même manière que ReLU pour des valeurs d'entrée positives. Cependant, pour des valeurs d'entrée négatives, ELU donne des valeurs lisses inférieures à 0.

$$\text{ELU}(x) = \begin{cases} x, & \text{si } x > 0 \\ \alpha(e^x - 1), & \text{si } x \leq 0 \end{cases}$$

avec $\alpha > 0$



Fonction ELU – Exponential Linear Unit

La fonction d'activation ELU a montré des résultats prometteurs dans diverses applications du Deep Learning. Cette alternative lissée à la fonction ReLU pallie certaines de ses limitations tout en conservant ses propriétés bénéfiques pour l'entraînement des réseaux de neurones.

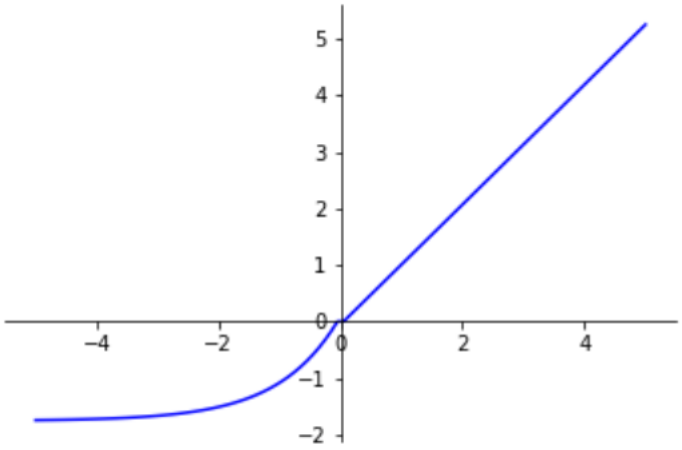
SELU

La fonction Scaled Exponential Linear Unit (SELU) est une amélioration de ELU.

SELU multiplie une variable scale au résultat de la fonction ELU. Cette opération a été conçue pour favoriser la normalisation des valeurs de sortie.

$$\text{SELU}(x) = \begin{cases} \lambda x, & \text{si } x > 0 \\ \lambda\alpha(e^x - 1), & \text{si } x \leq 0 \end{cases}$$

avec les constantes $\alpha \approx 1.67326324$, $\lambda \approx 1.05070098$



Fonction SELU – Scaled Exponential Linear Unit

Comme ELU, SELU pallie certaines des limitations de ReLU tout en conservant ses propriétés bénéfiques pour l'entraînement des réseaux de neurones.

CHAPITRE 2

CONSTRUCTION DE L'AUTOENCODER

2.1 Présentation de la base

L'AutoEncoder est appliqué à la base `base_Edu` qui contient :

— **366** variables après traitement : **111 continues** et **255 qualitatives**.

— Après application d'un *One-Hot Encoding*, la base atteint **717 variables**.

L'objectif est de tester différentes configurations d'AutoEncoder en variant certains hyperparamètres.

2.2 Expérimentations réalisées

2.2.1 Normalisation

Deux méthodes sont comparées :

— **StandardScaler** : Centrage autour de 0.

— **MinMaxScaler** : Transformation des valeurs entre 0 et 1.

Observation : La normalisation MinMax donne de meilleurs résultats.

2.2.2 Fonctions d'activation testées

Différentes combinaisons de fonctions d'activation sont expérimentées :

— **Encodeur** : Linear, Tanh, ReLU, Softsign, ELU, Sigmoid.

— **Décodeur** : Linear, Sigmoid.

Observation : La combinaison **Linear en encodeur** et **Sigmoid en décodeur** semble plus efficace.

2.2.3 Architecture du réseau

Tests réalisés sur :

— **Nombre de couches cachées** : 1 ou 2 couches.

— **Nombre de neurones par couche** : 150, 358, 500.

— **Fonction d'activation des couches cachées** : ReLU.

Observation : Une seule couche cachée avec ReLU est conservée pour la suite des tests.

2.3 Analyse des résultats

Configuration	Normalisation	Observation
Linear - Sigmoid (MinMax)	MinMax	Erreur de reconstruction plus faible
ReLU - Sigmoid (MinMax)	MinMax	Légère dégradation des performances
ELU - Sigmoid (MinMax)	MinMax	Moins performant que Linear-Sigmoid

TABLE 2.1 – Comparaison des combinaisons testées

2.4 Conclusion

Les expériences réalisées montrent que :

— La normalisation **MinMax** améliore la reconstruction des données.

— La combinaison **Linear en encodeur** et **Sigmoid en décodeur** est plus performante.

— Une architecture avec **une seule couche cachée activée par ReLU** est retenue.

.1 Annexe

Ici, l'objectif sera d'exploiter les éléments théoriques du chapitre 1 c'est à dire agir sur les fonctions d'activations pour la construction d'un AutoEncoder sur la base de données **base_Edu**.

Pour rappel, la base de données **base_Edu** contient après traitement 366 variables dont 111 variables continues et 255 qualitatives (catégorielles). Après un OneHotEncoder appliquée aux variables catégorielles, on s'est retrouvé à un total de 717 variables.

En général, l'objectif recherché lors de la construction d'un autoencoder est la minimsation de l'erreur de reconstruction. Dans notre cas, nous rechercherons plutôt un bon compromis entre une faible dimension de l'espace latent et une erreur minimisée.

On réalisera à chaque fois l'autoencoder pour les valeurs de la dimension latente allant de 1 à 20 pour retrouver la plus optimale bien-sûr en faisant varier quelques autres hyperparamètres.

De plus on utilisera le même nombre de couches cachés au niveau de l'encoder et du décodeur afin d'assurer une reconstruction efficace des données. Cette symétrie permet de préserver une complexité équivalente entre les deux parties du réseau, évitant ainsi qu'un encodeur trop profond n'extraie des représentations trop complexes que le décodeur aurait du mal à reconstruire. De plus, chaque couche de l'encodeur réduit progressivement la dimension des données, et une architecture symétrique garantit une expansion progressive dans le décodeur, facilitant ainsi l'apprentissage.

Une normalisation de type Standardisation c'est à dire un StandardScaler sera appliquée dans un premier temps. Nos données contiennent des valeurs négatives et une normalisation de type StandardScaler n'est pas censé rendre positive toute les valeurs.

Notre base de données comportant des valeurs négatives, voici les fonctions d'activations recommandées en dernière couche et pourquoi :

— Activation linéaire

  L'activation linéaire est le choix optimal lorsque les données normalisées conservent des valeurs négatives et positives. Cette activation permet de restituer l'intégralité de la plage des valeurs sans contrainte artificielle. Elle est particulièrement adaptée lorsque la normalisation des données a été réalisée avec un *StandardScaler*, qui centre les données autour de zéro.

— Tangente hyperbolique (Tanh) La fonction *Tanh* est une alternative possible. Elle contraint les valeurs dans l'intervalle $[-1, 1]$, ce qui peut être utile si les données normalisées restent dans cette plage. Cependant, si les valeurs initiales ont une dispersion plus large, cette activation risque de comprimer excessivement les sorties, entraînant une perte d'information. Cela n'est pas le cas ici. Nous ne pouvons donc pas l'utiliser en sortie dans le décoder.

Certaines fonctions d'activation ne conviennent pas lorsque les données d'entrées contiennent des valeurs négatives :

— **Sigmoïde** : Contraint les valeurs entre 0 et 1, ce qui est incompatible avec des données négatives.

— **ReLU et Leaky ReLU** : Suppriment ou modifient les valeurs négatives, faussant ainsi la reconstruction des données.

Nous nous servirons donc des fonctions d'activations **linear** et **tanh** pour construire l'autoencodeur avec plusieurs combinaisons possibles

TABLE 2 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	tanh	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	linear	717

Latent Dimension	**Mean Reconstruction Error**
1	0.159675
2	0.159673
3	0.147222
4	0.147234
5	0.132231
6	0.131572
7	0.125205
8	0.123194
9	0.119494
10	0.116373
11	0.111139
12	0.110259
13	0.106874
14	0.103370
15	0.102955
16	0.099901
17	0.099563
18	0.094729
19	0.093148
20	0.092328

La sortie précédente montre les erreurs de reconstruction obtenus pour les valeurs de la dimension latente allant de 1 à 20 pour la paramétrisation définie c'est à dire avec tanh en encodeur et linear en sortie. Le erreurs de reconstructions sont plus ou moins faible . Mais on préfèrera comparer cette sortie à celle d'une autre paramétrisation pour discuter son optimalité.

On choisit maintenant d'utiliser linear dans le décodeur et l'encodeur. tanh ne peut-être utilisé dans le décodeur. La reconstruction serait bien évidemment mauvaise puisque ses valeurs de sorties sont compris entre 0 et 1

TABLE 3 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	linear	717

Latent Dimension	**Mean Reconstruction Error**
1	0.159461
2	0.146873
3	0.136987
4	0.133950
5	0.129613
6	0.124726
7	0.120857
8	0.117318
9	0.113801
10	0.110772
11	0.107926
12	0.105275
13	0.102701
14	0.100242
15	0.097736
16	0.095122
17	0.092804
18	0.090603
19	0.088436
20	0.086282

La sortie précédente équivaut presque qu'a la précédente même si elle est légèrement meilleur. Mais cette amélioration n'est pas significative. Les deux paramétrisation s'équivalent presque Changeons donc le type de normalisation pour voir si on observera une amélioration significative des résultats. Reprenons les deux opérations précédentes mais cette fois ci avec une normalisation MinMax. Pour rappel, la normalisation MinMax rend toutes les valeurs positives et comprises entre 0 et 1. On pourra donc au besoin se servir de nouvelles fonction d'activation.

TABLE 4 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	tanh	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	linear	717

Latent Dimension	**Mean Reconstruction Error**
1	0.039125
2	0.039054
3	0.038350
4	0.037681
5	0.036700
6	0.036723
7	0.035105
8	0.035373
9	0.034300
10	0.034263
11	0.033511
12	0.033162
13	0.032183
14	0.031595
15	0.031692
16	0.030970
17	0.030797
18	0.029745
19	0.029150
20	0.029607

Les résulats obtenus sont nettement meilleurs par rapport aux précédentes. Les erreurs de reconstruction varient ici entre 0.029 et 0.039 alors que dans le cas avec la normalisation StandardScaler, elle varie entre 0.086 et 0.16.

TABLE 5 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	linear	717

Latent Dimension	**Mean Reconstruction Error**
1	0.038938
2	0.038771
3	0.037327
4	0.036199
5	0.035390
6	0.034759
7	0.034135
8	0.033567
9	0.032920
10	0.032356
11	0.031713
12	0.031168
13	0.030666
14	0.030144
15	0.029680
16	0.029184
17	0.028742
18	0.028266
19	0.027810
20	0.027415

Les résultats obtenus ici sont aussi meilleurs par rapport à ceux obtenus dans le cas de la normalisation StandardScaler. Elle est aussi légèrement meilleur par rapport à la sortie précédente. Tout cela témoigne du fait que la normalisation MinMax est beaucoup plus efficace par rapport à la normalisation StandardScaler. En effet la normalisation MinMax concentre les données entre 0 et 1. Il sont donc moins dispersé et donc certainement plus facile à reconstruire. Aussi la fonction linear semble plus efficace en encodeur que tanh.

Dans la suite on maintient donc la normalisation MinMax et on préfèrera la fonction d'activation linear à tanh en encodeur. Nous pouvons maintenant faire appel à un plus large éventail de fonction d'activations tel que sigmoid puisque les données sont maintenant positives compris entre 0 et 1, relu.

Dans le cas où les données sont compris entre 0 et 1 comme le notre, on fait souvent appel à la fonction sigmoid. Comparons-la à la fonction linear pour déterminer laquelle est la plus optimale et à quel poste.

TABLE 6 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	sigmoid	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.046330
2	0.042173
3	0.040817
4	0.040261
5	0.039628
6	0.039292
7	0.038816
8	0.037926
9	0.038441
10	0.037155
11	0.036642
12	0.036050
13	0.035551
14	0.034969
15	0.034521
16	0.034329
17	0.033858
18	0.033657
19	0.033229
20	0.032584

TABLE 7 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	sigmoid	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	linear	717

Latent Dimension	**Mean Reconstruction Error**
1	0.039600
2	0.039138
3	0.038589
4	0.038522
5	0.037473
6	0.037454
7	0.035908
8	0.036135
9	0.035234
10	0.034444
11	0.034099
12	0.032554
13	0.032139
14	0.031423
15	0.030750
16	0.030304
17	0.029782
18	0.029356
19	0.028753
20	0.028454

TABLE 8 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.040064
2	0.038473
3	0.036750
4	0.035542
5	0.034803
6	0.033912
7	0.033066
8	0.032408
9	0.031715
10	0.030936
11	0.030406
12	0.029602
13	0.029152
14	0.028528
15	0.028089
16	0.027475
17	0.026900
18	0.026290
19	0.025842
20	0.025448

La fonction linear semble donc au vu des résultats meilleurs au niveau de l'encoder et sigmoid au niveau du décodeur .

Lorsque l'on applique une normalisation MinMax, les observations sont transformées pour être comprises entre 0 et 1. Dans ce contexte, l'utilisation d'une activation linéaire dans l'encodeur et sigmoïde dans le décodeur peut s'expliquer par leurs propriétés respectives.

L'encodeur apprend une représentation latente des données. Une activation linéaire permet à l'encodeur de projeter les données sans restriction particulière sur l'échelle des valeurs, ce qui évite des effets de saturation prématurée et facilite l'apprentissage des caractéristiques pertinentes.

Le décodeur, quant à lui, doit reconstruire les données dans un intervalle [0,1], ce qui correspond parfaitement à la plage de sortie d'une fonction sigmoïde. En contraignant naturellement les sorties à rester dans cet intervalle, l'activation sigmoïde évite que les valeurs reconstruites dépassent les bornes imposées par la normalisation MinMax. Ainsi, cette combinaison linéaire/sigmoïde assure une reconstruction cohérente avec la distribution initiale des données tout en préservant l'information apprise par l'encodeur.

On préfera donc dans la suite de notre travail la fonction sigmoid en decodeur à la place de linear et linear en encodeur à la place de sigmoid.

Appliquons maintenant de nouvelles fonctions d'activations pour voir s'il est possible de faire mieux.

Commencons par la fonction d'activation relu.

TABLE 9 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	relu	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	relu	717

Latent Dimension	**Mean Reconstruction Error**
1	0.199781
2	0.133432
3	0.127316
4	0.086558
5	0.084759
6	0.071410
7	0.071034
8	0.076334
9	0.068217
10	0.069518
11	0.060692
12	0.059150
13	0.058437
14	0.056321
15	0.058398
16	0.060707
17	0.052819
18	0.058961
19	0.052364
20	0.052656

TABLE 10 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	relu	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.040058
2	0.038462
3	0.038434
4	0.037066
5	0.034762
6	0.034194
7	0.033959
8	0.034809
9	0.033339
10	0.032552
11	0.032506
12	0.030143
13	0.030654
14	0.030120
15	0.030233
16	0.028916
17	0.029229
18	0.027252
19	0.028849
20	0.027179

Les sorties précédentes indiquent que relu est meilleur en encodeur et sigmoid en decodeur par rapport à relu. Les résultat obtenus se rapprochent beaucoup plus de ceux obtenus avec linear en encodeur et sigmoid en decodeur. Cependant linear en encodeur employé avec sigmoid en décodeur reste toujours mieux.

Passons maintenant à une autre fonction d'activation.

Nous pouvons oublier mettre de côté la fonction d'activation softmax puisqu'elle transforme l'ensemble des observations d'une variable en distributions de probabilité. On obtiendrait certainement de mauvais résultats puisque cette structure est incompatible avec l'ensemble de nos observations.

On choisit donc de tester plutôt la fonction softsign

TABLE 11 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	softsign	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	softsign	717

Latent Dimension	**Mean Reconstruction Error**
1	0.074318
2	0.070148
3	0.052440
4	0.049915
5	0.048163
6	0.047148
7	0.045844
8	0.045189
9	0.044337
10	0.044100
11	0.043686
12	0.042819
13	0.042454
14	0.042162
15	0.042267
16	0.041624
17	0.041214
18	0.041824
19	0.040845
20	0.040616

TABLE 12 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	softsign	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.043954
2	0.042082
3	0.039981
4	0.039648
5	0.039283
6	0.039173
7	0.038918
8	0.038628
9	0.038210
10	0.037928
11	0.037801
12	0.037147
13	0.036870
14	0.036471
15	0.036389
16	0.035898
17	0.035729
18	0.035594
19	0.035114
20	0.034843

Les résultats obtenus avec softsign ne sont pas aussi bon que ceux obtenus avec linear en encodeur et sigmoid en decodeur.

Essayons une dernière fonction d'activation : elu

TABLE 13 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	elu	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	elu	717

Latent Dimension	**Mean Reconstruction Error**
1	0.038928
2	0.038779
3	0.037322
4	0.035851
5	0.035387
6	0.034712
7	0.034068
8	0.033502
9	0.033184
10	0.032280
11	0.031704
12	0.031212
13	0.030664
14	0.030165
15	0.029640
16	0.029205
17	0.028772
18	0.028355
19	0.027846
20	0.027445

.

TABLE 14 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
encoded	elu	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.040062
2	0.038442
3	0.037641
4	0.035850
5	0.035686
6	0.033910
7	0.033170
8	0.032394
9	0.032179
10	0.031109
11	0.030681
12	0.029799
13	0.029428
14	0.028837
15	0.028800
16	0.027616
17	0.027739
18	0.026801
19	0.026252
20	0.025614

Les résultats précédemment obtenus montre que la combinaison elu en encodeur et sigmoid en décodeur produit de meilleurs performance. Mais par rapport à la combinaison linear-sigmoid, elle ne fait pas le poids.

L'ensemble des résultats précédents semblent indiquer qu'il serait préférable de garder la fonction d'activation linear en encodeur et sigmoid en decodeur.

Nous allons maintenant agir sur un dernier hyperparamètre pour fixer notre autoencoder : Le nombre de couche(Les couches cachées) et le nombre de neurones par couches.

On commence donc par introduire une couche cachée en encodeur et au niveau du décodeur en utilisé comme nombre de neurones pour les couches cachées 358 égale à la moitité du nombre de variables (717).

Choisir 358 neurones dans une couche cachée d'un réseau de neurones, surtout lorsque l'on part de 717 variables d'entrée, est souvent basé sur une stratégie de réduction de la dimensionnalité tout en préservant l'information essentielle pour la reconstruction ou l'apprentissage. Ce choix est souvent motivé par la nécessité d'atteindre un compromis entre la capacité de modélisation du réseau et la précision de la

reconstruction des données. En réduisant le nombre de neurones à environ la moitié des variables d'entrée, cela permet au modèle d'apprendre des représentations compactes et généralisables des données tout en évitant le sur-apprentissage (overfitting) qui pourrait se produire si le réseau avait trop de neurones par rapport aux données disponibles. De plus, cette réduction facilite la simplification du modèle, ce qui peut améliorer l'efficacité computationnelle et l'interprétabilité du modèle, sans perdre d'informations clés. En résumé, choisir 358 neurones dans une couche cachée est une approche raisonnable pour équilibrer la capacité d'expression du réseau et les risques de complexité excessive, tout en permettant une bonne généralisation sur de nouvelles données.

Aussi nous choisissons d'utlliser comme fonction d'activation dans les couches cachées la fonction relu.

La fonction d'activation ReLU (Rectified Linear Unit) est largement préférée dans les couches cachées de l'encodeur et du décodeur des autoencodeurs, car elle résout efficacement le problème du vanishing gradient qui affecte d'autres fonctions d'activation comme sigmoïde et tanh. Ce problème peut ralentir considérablement l'entraînement et rendre l'optimisation difficile, notamment dans les réseaux profonds où la propagation des gradients à travers de nombreuses couches est cruciale. ReLU, en permettant une propagation stable et rapide des gradients pour les valeurs positives, améliore ainsi la convergence du modèle. De plus, sa simplicité et son efficacité computationnelle en font un choix optimal dans les architectures modernes, où les performances et la rapidité d'entraînement sont primordiales. Cette approche est largement validée par la littérature, notamment dans des travaux de Glorot et al. (2011), qui soulignent les avantages de ReLU dans les réseaux neuronaux profonds.

TABLE 15 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden	relu	358
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output	relu	358
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.037135
2	0.034914
3	0.033215
4	0.032070
5	0.031082
6	0.030195
7	0.029149
8	0.028904
9	0.027570
10	0.026772
11	0.025407
12	0.025277
13	0.024824
14	0.023571
15	0.023371
16	0.023213
17	0.022328
18	0.021138
19	0.021227
20	0.020941

Les résultats obtenus sont nettement meilleurs à ceux obtenus avec une seule couche en encodeur et au niveau du décodeur. Augmentons le nombre de couches pour voir si les résultats seront meilleurs.

TABLE 16 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden1	relu	358
Hidden2	relu	179
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output1	relu	179
Hidden_output2	relu	358
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.036781
2	0.034998
3	0.033751
4	0.032084
5	0.031624
6	0.031089
7	0.029956
8	0.029221
9	0.028453
10	0.028290
11	0.027788
12	0.027430
13	0.027134
14	0.026959
15	0.025263
16	0.024292
17	0.023944
18	0.023961
19	0.022658
20	0.022944

Les résultats précédents semblent indiqués que l'utilisation de plus de deux couches impact négativement les résultats. On choisit donc de se restreintre à une couche cachée dans l'encodeur et dans le décodeur.

On va maintenant agir sur le nombre de neurone des couches et voir si les résultats peuvent significativement être meilleur.

Couche(s) de l'encodeur	Fonction(s) d'activation	Nombre de neurones
Hidden	relu	500
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	Fonction(s) d'activation	Nombre de neurones
Hidden_output	relu	500
decoded	sigmoid	717

Latent Dimension	Mean Reconstruction Error
1	0.037140
2	0.034835
3	0.033543
4	0.032232
5	0.030888
6	0.030170
7	0.028998
8	0.028011
9	0.027650
10	0.026490
11	0.026118
12	0.024956
13	0.024814
14	0.023549
15	0.023234
16	0.023219
17	0.022049
18	0.022055
19	0.021797
20	0.020883

TABLE 17 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden	linear	150
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output	sigmoid	150
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.037199
2	0.035373
3	0.033980
4	0.032644
5	0.031440
6	0.030218
7	0.029684
8	0.028604
9	0.028129
10	0.027267
11	0.026187
12	0.025530
13	0.025237
14	0.024378
15	0.024206
16	0.023173
17	0.023013
18	0.022928
19	0.022126
20	0.021891

Pas d'amélioration significative observée pour une plus grand nombre de neurones et un plus petit nombre dans la couche cachée du décodeur et de l'encodeur.

On maintient donc pour notre autoencodeur la configuition suivante :

TABLE 18 – Évolution de la perte en fonction de la dimension de l'espace latent

Couche(s) de l'encodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden	relu	358
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output	relu	358
decoded	sigmoid	717

Latent Dimension	**Mean Reconstruction Error**
1	0.037135
2	0.034914
3	0.033215
4	0.032070
5	0.031082
6	0.030195
7	0.029149
8	0.028904
9	0.027570
10	0.026772
11	0.025407
12	0.025277
13	0.024824
14	0.023571
15	0.023371
16	0.023213
17	0.022328
18	0.021138
19	0.021227
20	0.020941