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.

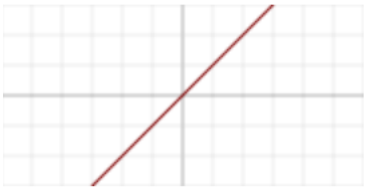
1.4.2 Exemples usuels de fontions d'activations

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

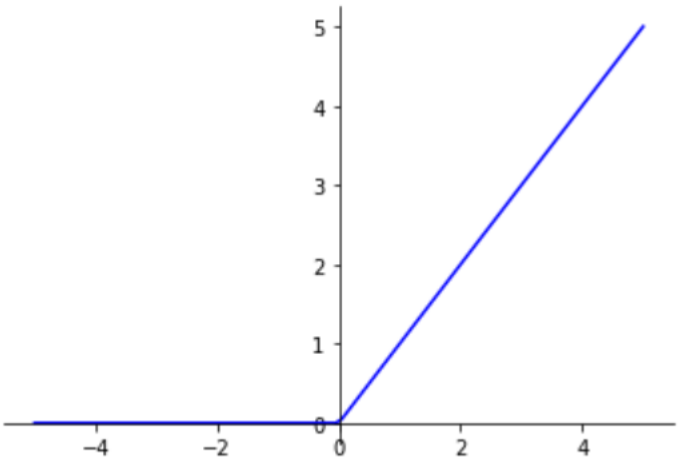
Utilisé en couche de sortie pour une utilisation pour une régression. On peut la caractériser de nulle, puisque les unités de sortie seront identiques à leur niveau d'entré. Intervalle de sortie $(-\infty; +\infty)$. Elle équivaut en faites à ne pas utiliser de fonction d'activation.



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.

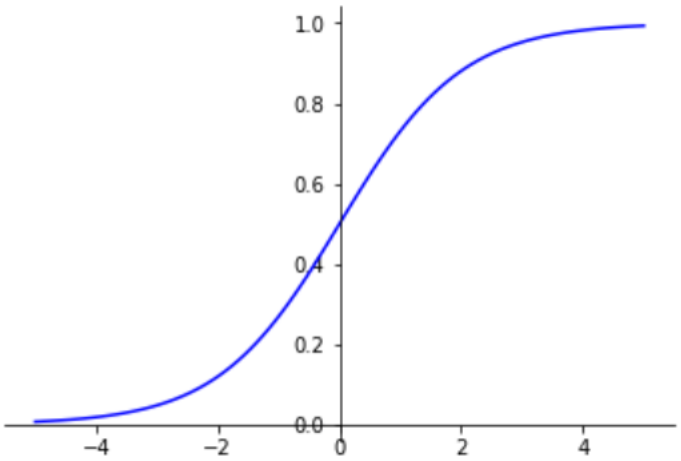
Cette faible pente pour les valeurs négatives permet aux neurones de continuer à apprendre même lorsque les entrées sont négatives, évitant ainsi la mort des neurones.

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.

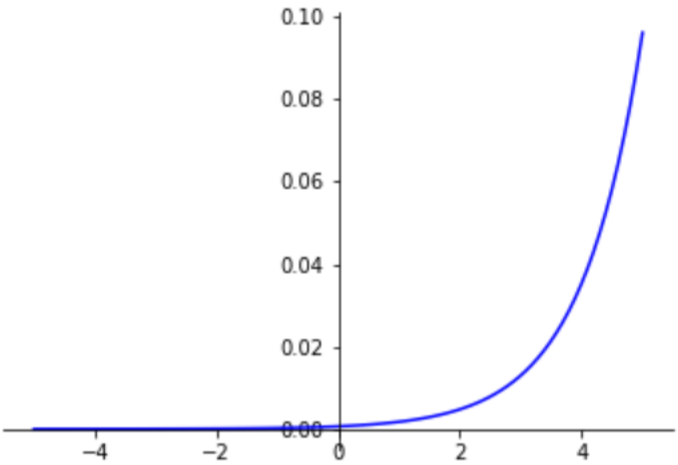
La fonction d'activation sigmoïde permet donc d'obtenir un résultat ambivalent, donnant une indication sur deux classes à la fois.

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.

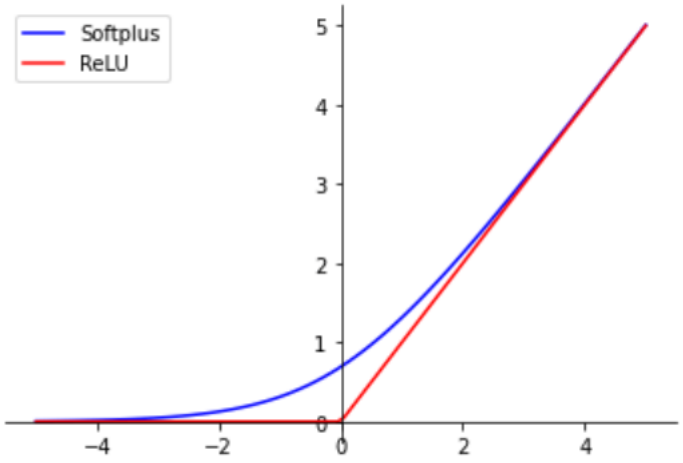
Softplus

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.

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



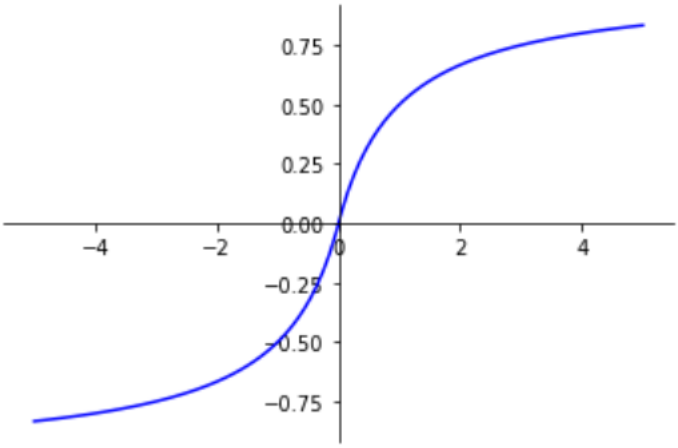
Fonction Softplus

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

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



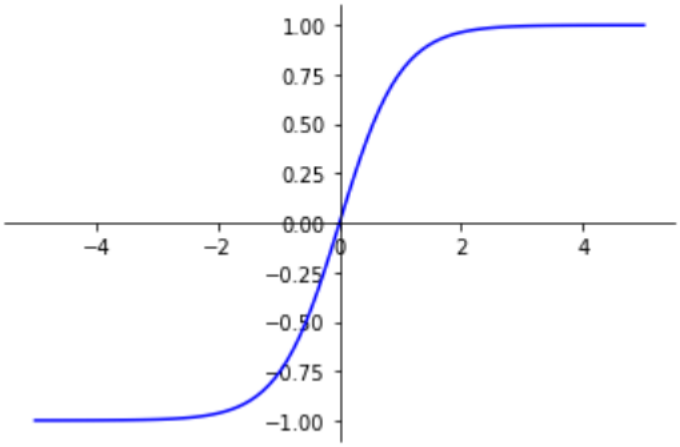
Fonction Softsign

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.

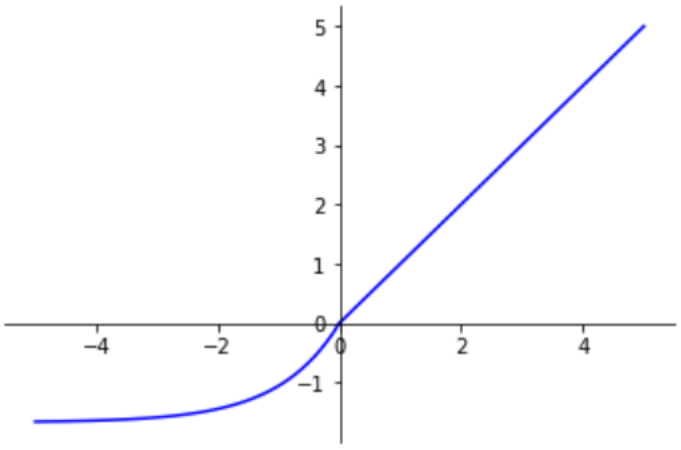
ELU

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.

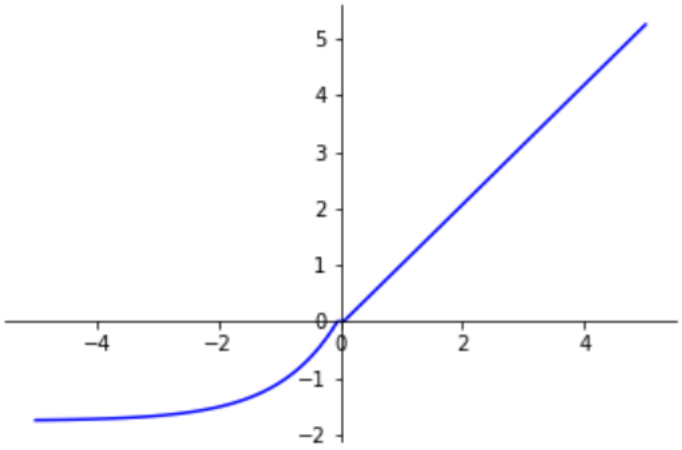
SELU

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 Construction de l'AutoEncodeur

Dans ce chapitre, nous mettons en application les principes théoriques définis précédemment afin de construire un autoencodeur optimisé pour la base de données **base_Edu**. L'objectif principal est d'obtenir un bon compromis entre la compression des données et la minimisation de l'erreur de reconstruction.

L'autoencodeur a été testé sous différentes configurations en faisant varier plusieurs hyperparamètres clés :

— **Normalisation des données** : Comparaison entre **StandardScaler** et **MinMax**, ce dernier donnant de meilleurs résultats.

— **Fonctions d'activation** : Test de plusieurs combinaisons (`Linear`, `Sigmoid`, `Tanh`, `ReLU`, `ELU`, `Softsign`), avec une préférence finale pour **Linear en encodeur** et **Sigmoid en décodeur**.

— **Architecture du réseau** : Analyse de l'impact du nombre de couches cachées et de la taille de l'espace latent, avec une meilleure performance observée pour une **architecture symétrique à une couche cachée de 350 neurones**.

L'ensemble des expérimentations et des résultats obtenus sont détaillés dans l'**Annexe**. Ces analyses permettent de justifier les choix retenus pour la construction finale de l'autoencodeur.

TABLE 2.1 – Comparaison des architectures d'AutoEncodeur

Approche	**Normalisation**	**Activation (Enc-Dec)**	**Perte min.**	**Perte max.**
Tanh-Linear	StandardScaler	Tanh - Linear	0.0929	0.1656
Linear-Linear	StandardScaler	Linear - Linear	0.0862	0.1651
Tanh-Linear	MinMax	Tanh - Linear	0.0299	0.0395
Linear-Linear	MinMax	Linear - Linear	0.0276	0.0394
Linear-Sigmoid	MinMax	Linear - Sigmoid	0.0256	0.0405
ELU-Sigmoid	MinMax	ELU - Sigmoid	0.0259	0.0405
Avec couche(s) cachée(s)				
Linear-Sigmoid (1 couche 350)	MinMax	Linear - Sigmoid	0.0237	0.0379
Linear-Sigmoid (2 couches 350-100)	MinMax	Linear - Sigmoid	0.0252	0.0378
Linear-Sigmoid (3 couches 350-100-50)	MinMax	Linear - Sigmoid	0.0343	0.0381
Linear-Sigmoid (1 couche 500)	MinMax	Linear - Sigmoid	0.0233	0.0380
Linear-Sigmoid (1 couche 150)	MinMax	Linear - Sigmoid	0.0247	0.0381
Approche retenue	**MinMax**	**Linear - Sigmoid (1 couche 350)**	**0.0237**	**0.0379**

CHAPITRE 3

APPLICATION AUX VARIABLES CONTINUES

Les erreurs de reconstructions donnent avec les variables continues

TABLE 3.1 – Évolution de la perte en fonction de la dimension de l'espace latent

encoding_dim	**loss**
1	0.012035
2	0.009823
3	0.007376
4	0.006276
5	0.005308
6	0.004686
7	0.004260
8	0.003889
9	0.003481
10	0.003224
11	0.002994
12	0.002785
13	0.002567
14	0.002426
15	0.002196
16	0.002057
17	0.001917
18	0.001768
19	0.001707
20	0.001650

On fera une comparaison avec l'ACP c'est à dire l'erreur de reconstruction obtenu en ACP

.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 373 variables dont 118 variables continues et 255 qualitatives (catégorielles). Après un OneHotEncoder appliquée aux variables catégorielles, on s'est retrouvé à un total de 724 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	724

encoding_dim	**loss**
1	0.165638
2	0.165620
3	0.153498
4	0.143808
5	0.143322
6	0.135287
7	0.126833
8	0.123304
9	0.121459
10	0.116878
11	0.114194
12	0.110568
13	0.109627
14	0.105574
15	0.103984
16	0.102650
17	0.100142
18	0.099032
19	0.094375
20	0.092898

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	724

encoding_dim	**loss**
1	0.165086
2	0.152853
3	0.142074
4	0.131576
5	0.130224
6	0.124636
7	0.120063
8	0.116461
9	0.113201
10	0.109978
11	0.106967
12	0.104291
13	0.101278
14	0.100131
15	0.097408
16	0.094958
17	0.092468
18	0.090250
19	0.088195
20	0.086154

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	724

encoding_dim	**loss**
1	0.039556
2	0.039483
3	0.039060
4	0.038396
5	0.038092
6	0.036511
7	0.035947
8	0.035372
9	0.035190
10	0.034430
11	0.033913
12	0.032952
13	0.033326
14	0.031847
15	0.031795
16	0.031376
17	0.031050
18	0.030250
19	0.029215
20	0.029971

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	724

encoding_dim	**loss**
1	0.039400
2	0.039207
3	0.037937
4	0.036849
5	0.035726
6	0.035077
7	0.034487
8	0.033838
9	0.033281
10	0.032614
11	0.032040
12	0.031539
13	0.030949
14	0.030439
15	0.030007
16	0.029486
17	0.029071
18	0.028529
19	0.028156
20	0.027627

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 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	724

encoding_dim	**loss**
1	0.046860
2	0.042649
3	0.041317
4	0.040714
5	0.040112
6	0.039719
7	0.039265
8	0.038183
9	0.038718
10	0.037491
11	0.037108
12	0.036485
13	0.035757
14	0.035382
15	0.034787
16	0.034793
17	0.034127
18	0.033638
19	0.032996
20	0.032576

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	724

encoding_dim	**loss**
1	0.040074
2	0.039631
3	0.039070
4	0.039073
5	0.037984
6	0.037971
7	0.035767
8	0.035957
9	0.035251
10	0.034412
11	0.034297
12	0.033521
13	0.032896
14	0.031730
15	0.031284
16	0.030574
17	0.030212
18	0.029511
19	0.029345
20	0.028838

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	724

encoding_dim	**loss**
1	0.040540
2	0.038941
3	0.037166
4	0.035834
5	0.034982
6	0.034186
7	0.033445
8	0.032645
9	0.031944
10	0.031316
11	0.030590
12	0.030090
13	0.029473
14	0.028763
15	0.028340
16	0.027626
17	0.027136
18	0.026565
19	0.026089
20	0.025598

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	724

encoding_dim	**loss**
1	0.198739
2	0.145789
3	0.110483
4	0.083418
5	0.075333
6	0.071814
7	0.067407
8	0.063470
9	0.055202
10	0.061518
11	0.053417
12	0.048605
13	0.050918
14	0.047460
15	0.058121
16	0.047480
17	0.041547
18	0.047158
19	0.046316
20	0.044657

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	724

encoding_dim	**loss**
1	0.040533
2	0.040535
3	0.037505
4	0.036196
5	0.037455
6	0.035135
7	0.034327
8	0.033581
9	0.033477
10	0.032186
11	0.032851
12	0.030877
13	0.030258
14	0.029785
15	0.028998
16	0.029543
17	0.027949
18	0.029062
19	0.027489
20	0.026338

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	724

encoding_dim	**loss**
1	0.074998
2	0.062875
3	0.054187
4	0.050231
5	0.048522
6	0.047538
7	0.046178
8	0.045629
9	0.045143
10	0.044336
11	0.044023
12	0.043759
13	0.043124
14	0.042956
15	0.042323
16	0.041676
17	0.041556
18	0.041489
19	0.041142
20	0.040880

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	724

encoding_dim	**loss**
1	0.044685
2	0.042594
3	0.040434
4	0.040156
5	0.039713
6	0.039556
7	0.039377
8	0.039038
9	0.038651
10	0.038315
11	0.038165
12	0.037675
13	0.037334
14	0.037011
15	0.036565
16	0.036727
17	0.035852
18	0.035729
19	0.035508
20	0.035196

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	724

encoding_dim	**loss**
1	0.039390
2	0.039184
3	0.037778
4	0.036732
5	0.035886
6	0.035085
7	0.034433
8	0.033809
9	0.033313
10	0.032815
11	0.032012
12	0.031553
13	0.031004
14	0.030462
15	0.030046
16	0.029492
17	0.028965
18	0.028582
19	0.028115
20	0.027637

.

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	724

encoding_dim	**loss**
1	0.040537
2	0.038563
3	0.037426
4	0.036116
5	0.035849
6	0.034410
7	0.033584
8	0.032830
9	0.033422
10	0.031742
11	0.031609
12	0.030276
13	0.029692
14	0.029240
15	0.028533
16	0.027843
17	0.027377
18	0.026832
19	0.026302
20	0.025930

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 maintenant sur un dernière 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 350.

Étant donné que l'entrée de l'autoencodeur comporte 724 variables après encodage one-hot des variables qualitatives, il est pertinent de choisir une couche cachée avec un nombre de neurones inférieur afin de capturer les structures latentes tout en réduisant la dimensionnalité. Un choix de 350 neurones permet un bon compromis entre compression et préservation de l'information : il est suffisamment élevé pour éviter une perte excessive d'information mais réduit efficacement la redondance présente dans les

données initiales. De plus, cette réduction progressive facilite l'apprentissage des représentations latentes tout en limitant le risque de surapprentissage.

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	linear	350
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output	sigmoid	350
decoded	sigmoid	724

encoding_dim	**loss**
1	0.037949
2	0.036440
3	0.034920
4	0.034126
5	0.032892
6	0.031767
7	0.030695
8	0.030134
9	0.029405
10	0.028889
11	0.027681
12	0.027590
13	0.026882
14	0.026415
15	0.026250
16	0.025424
17	0.025097
18	0.024562
19	0.024317
20	0.023735

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	linear	350
Hidden2	linear	100
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output1	sigmoid	100
Hidden_output2	sigmoid	350
decoded	sigmoid	724

encoding_dim	**loss**
1	0.037874
2	0.037458
3	0.035550
4	0.034632
5	0.033438
6	0.032716
7	0.031777
8	0.031131
9	0.030446
10	0.029779
11	0.029180
12	0.028650
13	0.028151
14	0.028040
15	0.026990
16	0.026858
17	0.026389
18	0.026262
19	0.025733
20	0.025273

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**
Hidden1	linear	350
Hidden2	linear	100
Hidden3	linear	50
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output1	sigmoid	50
Hidden_output2	sigmoid	100
Hidden_output3	sigmoid	350
decoded	sigmoid	724

encoding_dim	**loss**
1	0.037775
2	0.038126
3	0.035760
4	0.035160
5	0.035260
6	0.034805
7	0.034765
8	0.034896
9	0.035552
10	0.035073
11	0.034401
12	0.034124
13	0.034946
14	0.034630
15	0.034730
16	0.035156
17	0.034732
18	0.034272
19	0.034449
20	0.034307

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	linear	500
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	Fonction(s) d'activation	Nombre de neurones
Hidden_output	sigmoid	500
decoded	sigmoid	724

encoding_dim	loss
1	0.038024
2	0.036253
3	0.034950
4	0.033529
5	0.032815
6	0.031313
7	0.030745
8	0.029663
9	0.028687
10	0.027671
11	0.027572
12	0.026850
13	0.026562
14	0.025958
15	0.025551
16	0.024844
17	0.024388
18	0.024028
19	0.023159
20	0.023331

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	linear	150
encoded	linear	$k \in [1, 20]$

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

encoding_dim	**loss**
1	0.038111
2	0.037133
3	0.035719
4	0.034221
5	0.033173
6	0.032157
7	0.031188
8	0.030395
9	0.029601
10	0.029317
11	0.028778
12	0.028154
13	0.027947
14	0.027264
15	0.026820
16	0.026354
17	0.025865
18	0.025593
19	0.024790
20	0.024737

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 19 – 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	350
encoded	linear	$k \in [1, 20]$

Couche(s) du décodeur	**Fonction(s) d'activation**	**Nombre de neurones**
Hidden_output	sigmoid	350
decoded	sigmoid	724

encoding_dim	**loss**
1	0.037949
2	0.036440
3	0.034920
4	0.034126
5	0.032892
6	0.031767
7	0.030695
8	0.030134
9	0.029405
10	0.028889
11	0.027681
12	0.027590
13	0.026882
14	0.026415
15	0.026250
16	0.025424
17	0.025097
18	0.024562
19	0.024317
20	0.023735