

Compte-rendu de Travaux Pratiques de Intelligence Artificielle et Optimisations

Quentin DAVIN

Joachim DRUHET

5/12/2025

Table des matières

| | |
|---|-----------|
| TD 1 - Apprentissage | 3 |
| Introduction | 3 |
| Compréhension du programme | 3 |
| Entraînement du programme de la porte AND | 4 |
| Visualisation de l'évolution | 5 |
| Test des différents hyper-paramètres | 6 |
| coeff faible | 7 |
| coeff fort | 9 |
| iterations faible | 11 |
| Conclusion | 11 |
| Modification pour une porte OR | 13 |
| Modification pour une porte XOR | 15 |
| Conclusion | 17 |
| TD 2 - GAN | 19 |
| Introduction | 19 |
| GAN Lab | 19 |
| Analyse des hyper-paramètres | 19 |
| StyleGAN 3 | 21 |
| Instanciation du modèle | 21 |
| Conclusion | 22 |
| TD 3 - Transformers | 26 |
| Introduction | 26 |
| Préparation de l'environnement et des données | 26 |
| Installation et authentification | 26 |
| Dataset | 26 |
| Causal Language Model | 26 |
| Principe | 26 |
| Configuration | 26 |
| Entraînement | 26 |
| Masked Language Modeling (MLM) | 27 |
| Principe | 27 |
| Configuration | 27 |
| Entraînement | 27 |
| Analyse | 28 |
| Conclusion | 28 |

Table des Figures

| | | |
|----|--|----|
| 1 | Evolution de l'erreur en fonction du nombre d'itérations avec la porte AND | 4 |
| 2 | Evolution de l'erreur en fonction du nombre d'itération avec le paramètre <code>coeff</code> à 0,01 | 7 |
| 3 | Evolution des poids en fonction du nombre d'itération avec le paramètre <code>coeff</code> à 0,01 | 8 |
| 4 | Evolution de l'erreur en fonction du nombre d'itération avec le paramètre <code>coeff</code> à 10,0 | 9 |
| 5 | Evolution des poids en fonction du nombre d'itération avec le paramètre <code>coeff</code> à 10,0 | 10 |
| 6 | Evolution de l'erreur en fonction du nombre d'itération avec le paramètre <code>iterations</code> à 50 | 11 |
| 7 | Evolution des poids en fonction du nombre d'itération avec le paramètre <code>iterations</code> à 50 | 12 |
| 8 | Evolution de l'erreur en fonction du nombre d'itérations avec la porte OR | 13 |
| 9 | Evolution des poids en fonction du nombre d'itérations avec la porte OR | 14 |
| 10 | Evolution de l'erreur en fonction du nombre d'itérations avec la porte XOR | 15 |
| 11 | Evolution des poids en fonction du nombre d'itérations avec la porte XOR | 16 |
| 12 | Représentation graphique du problème avec les solutions | 18 |
| 13 | Génération avec une distribution gaussienne 1D | 20 |
| 14 | Image générée par StyleGAN3 avec la graine 2 | 23 |
| 15 | Image générée par StyleGAN3 avec la graine 674 | 24 |
| 16 | Image générée par StyleGAN3 avec la graine 1024 | 25 |

TD 1 - Apprentissage

Introduction

L'objectif de ce travail est d'illustrer le fonctionnement d'un réseau de neurones minimalistes (Perceptron) entraîné pour modéliser des fonctions logiques (AND, OR, XOR).

Compréhension du programme

Le programme `perceptron_learn.py` implémente un réseau de neurones artificiels élémentaire constitué d'un seul neurone (Perceptron). Son but est d'apprendre à reproduire une fonction logique binaire (par défaut la porte AND) via un algorithme d'apprentissage supervisé.

Dans ce code, la première section est la définition des entrées et des hyperparamètres:

Entrées

- `inputValue` : c'est la liste des entrées possibles ([00, 01, 10, 11])
- `numIn` : c'est le nombre d'entrées possibles
- `desired_out` : c'est la liste des sorties attendues par rapport aux entrées possibles. La configuration de base est la fonction logique AND.

Hyperparamètres

- `weights` : Ce sont les poids des neurones (leur valeur initialisée aléatoirement puis modifiée par les algorithmes.)
- `bias` : c'est le biais. Il permet de changer les valeurs de sortie des neurones pour pas être bloquer dans un état spécifique
- `coeff` : c'est le coefficient d'apprentissage. Il détermine la vitesse à laquelle les poids sont modifiés à chaque erreur (pas de gradient)
- `iterations` : c'est le nombre d'itérations

Itérations

Pour chaque itération :

- on crée une liste de valeurs de sorties remplies de 0

On itère les entrées possibles, pour chaque entrée, on fait :

- on calcule y comme la somme pondérée des entrées par les poids.
- puis on calcule la sortie, qui est le résultat de la fonction logistique $\frac{1}{1 + e^{-y}}$ à y .
- ensuite, on calcule l'erreur (`delta` dans le programme) : on soustrait la sortie désirée à la sortie obtenue
- finalement, on corrige les poids (répropagation) avec cette formule $W_{nouveau} = W_{actuel} + (coeff \times input \times delta)$.

Sorties A la fin du programme, il trace l'évolution de l'erreur au fil des itérations pour visualiser la convergence. De plus, il renvoie les poids finaux dans la console et dans un fichier `weights.csv`

Entraînement du programme de la porte AND

On lance le programme et on se retrouve avec ses sorties :

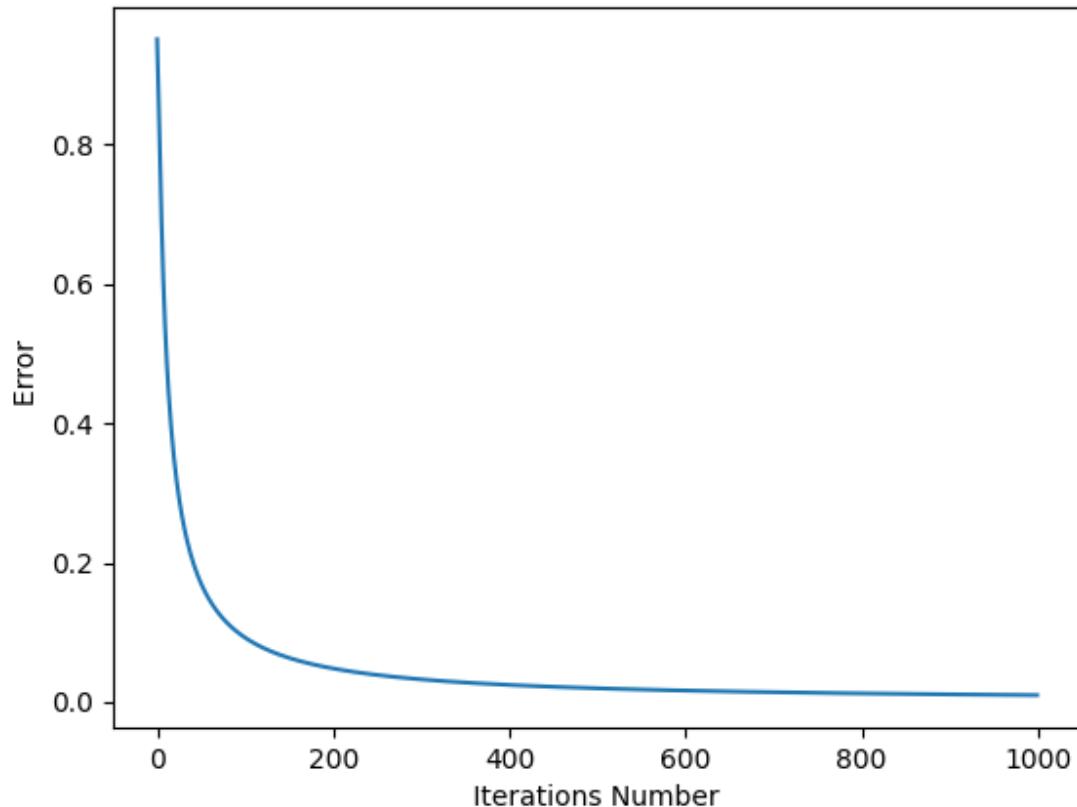


Figure 1: Evolution de l'erreur en fonction du nombre d'itérations avec la porte AND

On observe une réduction exponentielle des erreurs, qui se stabilise à, environ, 0,01 à l'itération n°400.
Concernant les poids, on se retrouve avec :

- $w_0 = 14.496149326345051$
- $w_1 = 9.56174144292132$
- $w_2 = 9.55552021068763$

On observe que $w_1 \simeq w_2$ alors que w_0 est 1,55 fois supérieur des autres poids.

Pour analyser ces poids, faut reprendre la formule de y .

$$y = (\text{bias} \times w_0) + (x_1 \times w_1) + (x_2 \times w_2)$$

Il y a trois cas possibles :

- Aucune entrée active (0,0) :

$$y = -14.5 + 0 + 0 = -14.5$$

Vu que le résultat est négatif, alors la sortie est 0

- Une seule entrée active (0,1) ou (1,0) :

$$y = -14.5 + 9.56 = -4.96$$

Malgré le fait qu'on ajoute le poids positif d'une entrée, le résultat reste négatif, donc la sortie vaut 0.

- Les deux entrées actives (1, 1):

$$y = -14.5 + 9.56 + 9.56 = 4.62$$

La somme des deux poids positifs dépasse le seuil du biais, ainsi le résultat devient positif, donc la sortie vaut 1.

Ainsi, les sorties correspondant bien à une porte AND.

Les poids w_1 et w_2 ont quasiment la même valeur car, dans la porte AND, les entrées x_1 et x_2 ont la même importance.

Visualisation de l'évolution

Pour visualiser l'évolution, on a réalisé des petites modifications dans le code.

Le premier est l'ajout de liste pour créer un historique :

```
w0_history: list[float] = list()
w1_history: list[float] = list()
w2_history: list[float] = list()
```

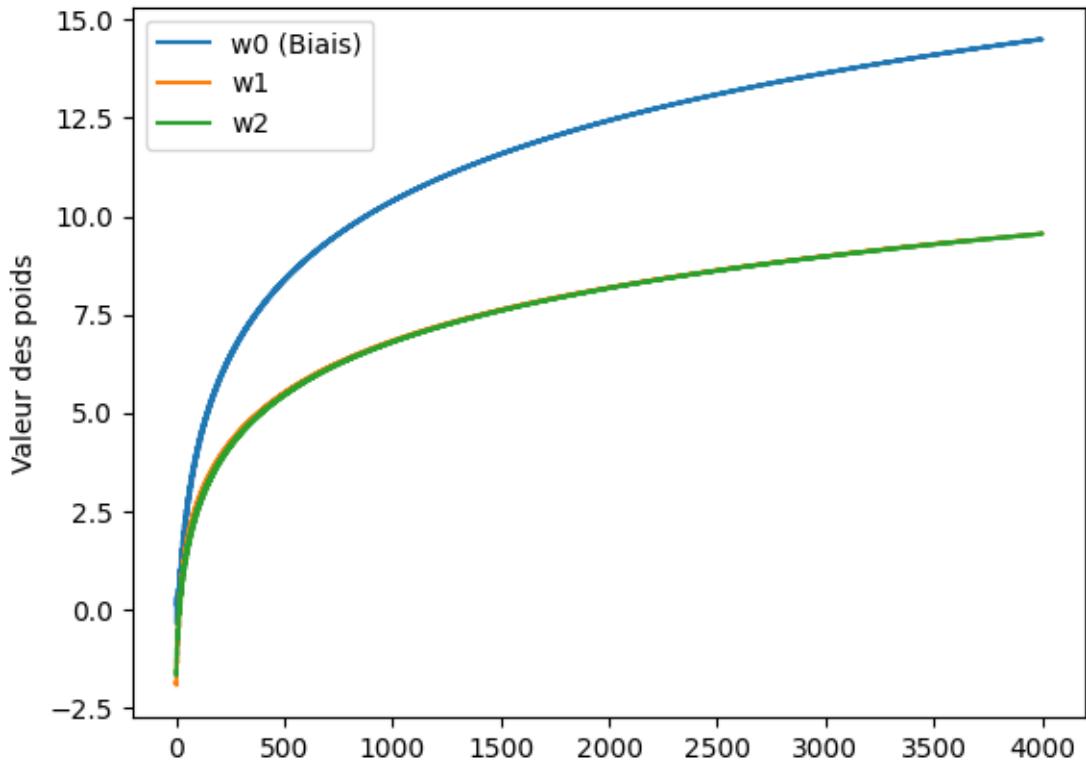
Le deuxième est l'ajout des valeurs de poids dans les listes :

```
w0_history.append(weights[0][0])
w1_history.append(weights[1][0])
w2_history.append(weights[2][0])
```

Le dernier est le tracage des trois historiques dans un même graphique :

```
plt.figure()
plt.plot(w0_history, label="w0 (Biais)")
plt.plot(w1_history, label="w1")
plt.plot(w2_history, label="w2")
plt.ylabel("Valeur des poids")
plt.legend()
plt.show()
```

Avec ces modifications, on se retrouve avec ce graphique pour la porte AND :



Dans ce graphique, on retrouve :

- La convergence rapide au début : on observe une augmentation très rapide des valeurs des poids dans les 200 premières itérations. C'est la phase où les erreurs sont encore grandes et où le réseau "apprend" le plus vite.
- La symétrie parfaite entre w_0 et w_1 : Les courbes orange (w_1) et verte (w_2) sont quasiment superposées. Cela confirme que pour une porte logique AND, les deux entrées jouent un rôle symétrique et ont exactement la même importance. Elles atteignent environ 9,5 à la fin.
- La dominance du biais (w_0) : la courbe bleue (w_0) reste constamment au-dessus des deux autres. Elle atteint environ 14.5. cela valide la condition logique de la porte AND : le poids du biais (qui agit comme seuil négatif car le biais vaut -1) doit être fort que w_1 ou w_2 individuellement pour empêcher l'activation si une seule entrée est à 1.
- L'absence de stabilisation totale : On remarque que les courbes ne sont pas parfaitement plates même après 1000 itérations : elles continuent de monter très doucement. Cela s'explique car les données sont parfaitement séparables, l'algorithme cherche à réduire l'erreur à zéro absolu. Avec une fonction sigmoïde, pour obtenir une sortie de exactement 1,0 (et non 0,999), l'entrée pondérée doit tendre vers l'infini. Les poids continuent donc de croître lentement pour "pousser" la sigmoïde vers ses extrêmes.

Test des différents hyper-paramètres

Dans ce programme, nous avons deux hyper-paramètres : `coeff` et `iterations`. Pour tester l'effet de ces hyper-paramètres sur ce réseau neuronal, on va réaliser ces expériences :

- `coeff` faible (par exemple, $coeff = 0,01$)

- `coeff` élevé (par exemple, $coeff = 10,0$)
- `iterations` faible (par exemple, $iterations = 50$)

coeff faible

Dans cette expérience, on va mettre l'hyper-paramètre `coeff` à 0,01.

Avec cette valeur, on trouve ce graphique d'évolution de l'erreur :

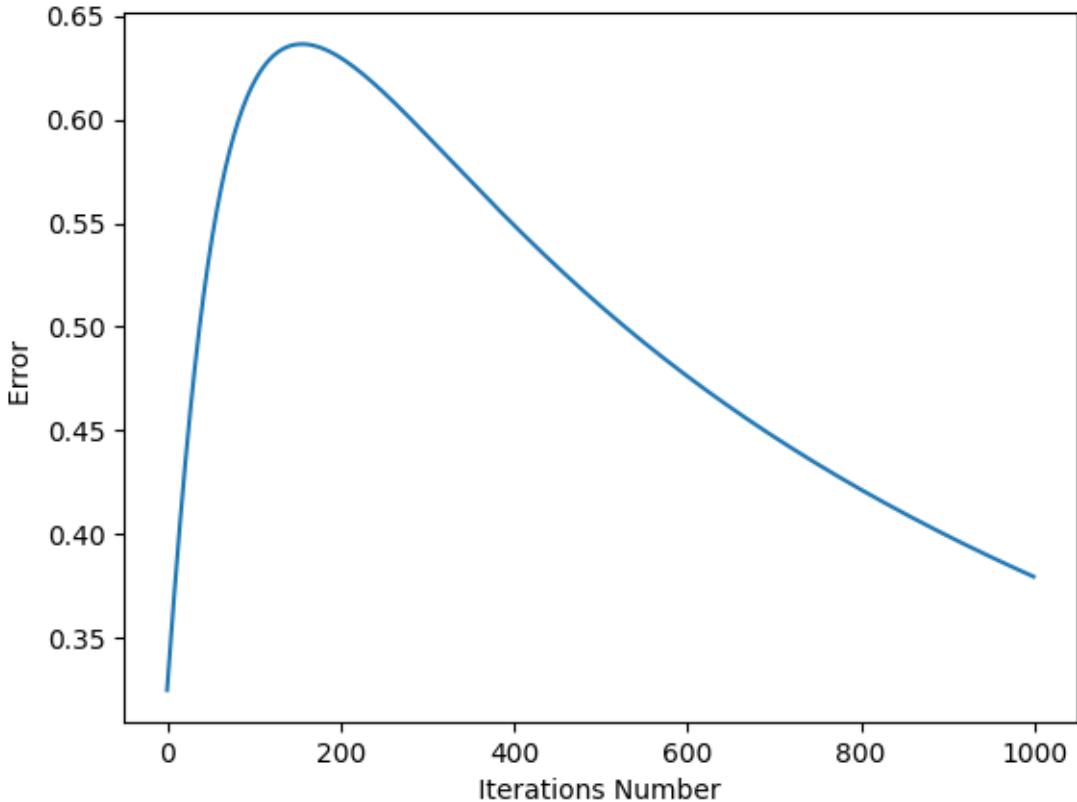


Figure 2: Evolution de l'erreur en fonction du nombre d'itération avec le paramètre `coeff` à 0,01

Et ce graphique d'évolution des poids :

On peut relever plusieurs observations de deux graphiques :

- Non-convergence de l'erreur : Contrairement au cas précédent, la courbe d'erreur ne descend pas vers 0. Après une phase initiale étrange où l'erreur augmente (jusqu'à, environ, 0,64), elle redescend très lentement mais stagne très lentement mais stagne encore à 0,38 à la fin des 1000 itérations. Le réseau n'a pas appris la logique.
- L'évolution insuffisante des poids : Sur le graphique des poids, on voit que les courbes sont linéaires et montantes, ce qui montre que la direction prise est la bonne (w_0 au-dessus de w_1 et w_2). Cependant, les valeurs finales sont ridicules : $w_0 \approx 2,7$ et $w_1/w_2 \approx 1,6$

Ainsi, avec $coeff = 0,01$, les pas effectués par l'algorithme de descente de gradient sont minuscules. Le réseau “apprend”, mais à une vitesse très lente. Pour attendre la convergence avec ce coefficient, il faudrait augmenter drastiquement le nombre d'itérations. Ici, l'algorithme s'est arrêté en plein milieu du chemin : c'est un cas des sous-apprentissage dû à une convergence trop lente.

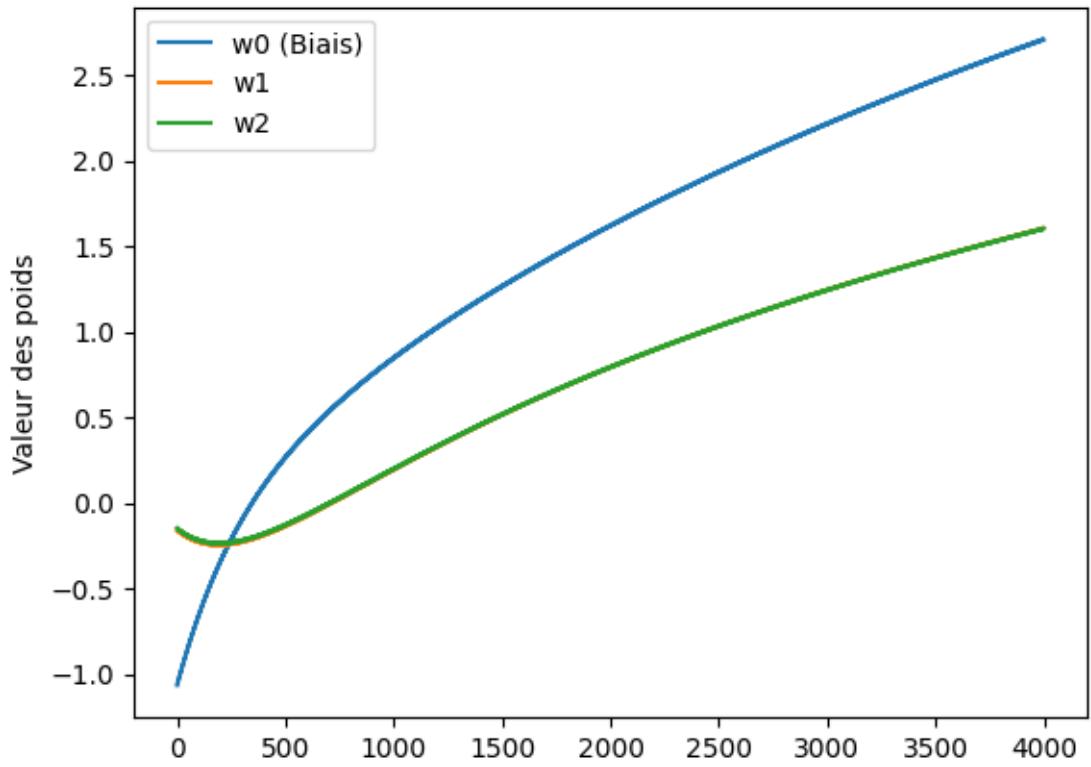


Figure 3: Evolution des poids en fonction du nombre d'itération avec le paramètre `coeff` à 0,01

coeff fort

Dans cette expérience, on va mettre l'hyper-paramètre **coeff** à 10,0.

Avec cette valeur, on trouve ce graphique d'évolution de l'erreur :

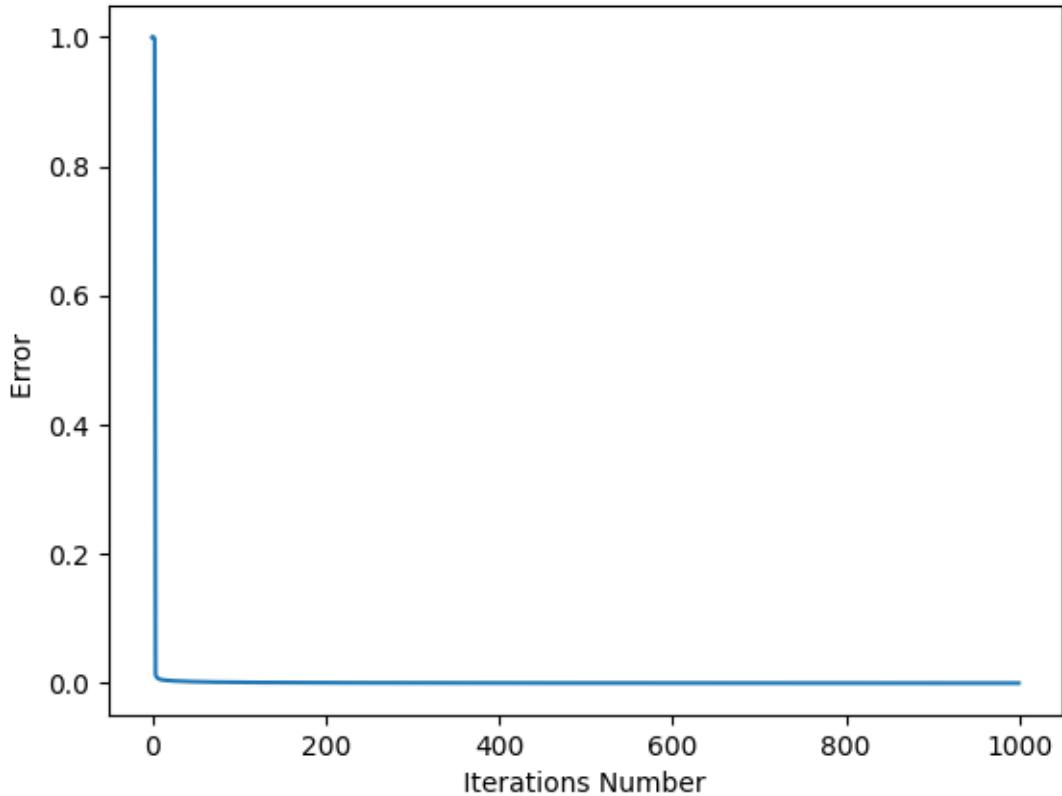


Figure 4: Evolution de l'erreur en fonction du nombre d'itération avec le paramètre **coeff** à 10,0

Et ce graphique d'évolution des poids :

A partir de ces deux graphiques, on peut tirer ces observations :

- Chute immédiate de l'erreur : Sur le graphique de l'erreur, la courbe tombe verticalement de 1.0 à 0 en moins de 10 itérations. Le réseau a trouvé une solution instantanément.
- Explosion initiale des poids : Sur le graphique des poids, on voit une ligne verticale au tout début (itération 0). Les poids ont bondi de 0 à 20/25 d'un seul coup. Cela est un effet "Bélier" : la première correction d'erreur provoque un déplacement gigantesque dans l'espace des poids. L'algorithme ne "descend" pas la pente doucement, il saute directement très loin.
- Perte de symétrie ($w_1 \neq w_2$) : Contrairement au cas "normal" ($coeff = 0,7$) où les courbes orange et verte étaient superposées, ici elles sont séparées ($w_1 \approx 18$ et $w_2 \approx 16$). Cela est dû au coefficient qui est trop fort. Lors de la première itération, les poids se voient modifié et cette modification est tellement forte, que les poids ne sont plus symétriques.

Cette configuration présente un risque car, malgré le fait que l'erreur soit nulle ici (dû à la facilité du problème), mais sur un problème plus complexe, ce coefficient ferait osciller l'algorithme sans jamais se stabiliser, ou faire diviser les poids vers l'infini (phénomène appelé "Exploding Gradient").

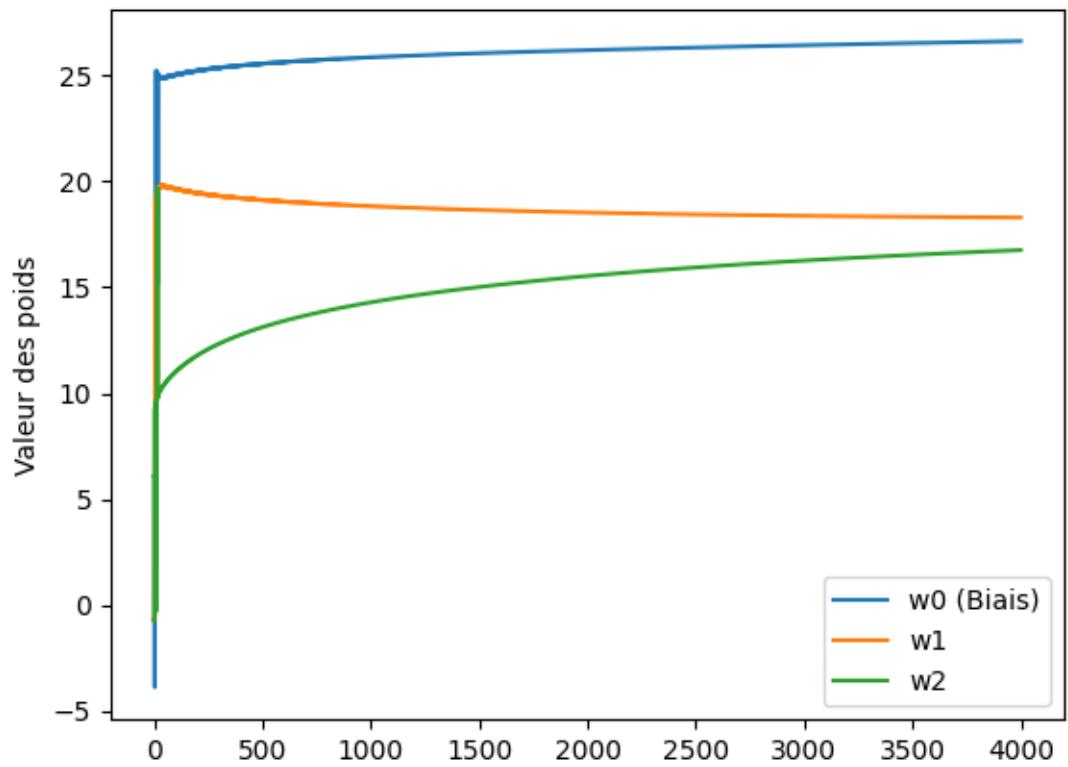


Figure 5: Evolution des poids en fonction du nombre d'itération avec le paramètre `coeff` à 10,0

iterations faible

Dans cette expérience, on va mettre l'hyper-paramètre `iterations` à 50.

Avec cette valeur, on trouve ce graphique d'évolution de l'erreur :

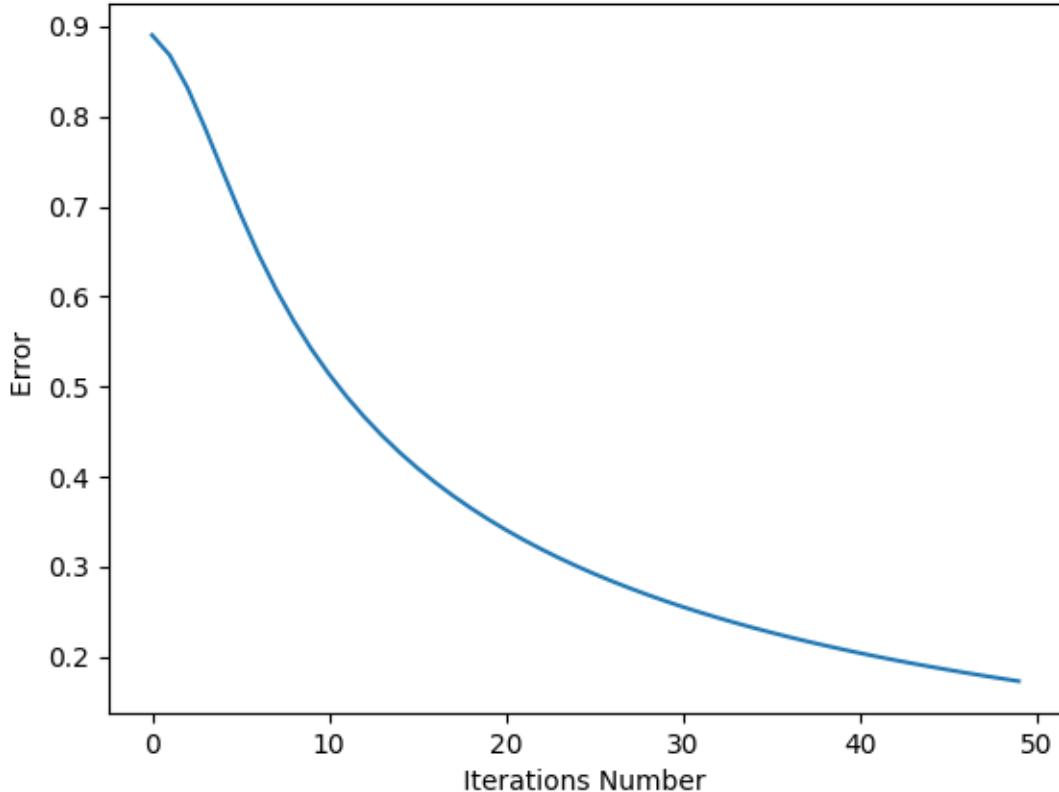


Figure 6: Evolution de l'erreur en fonction du nombre d'itération avec le paramètre `iterations` à 50

Et ce graphique d'évolution des poids :

A partir de ces deux graphiques, on peut tirer ces observations :

- Erreur non nulle : La courbe d'erreur descend proprement, mais s'arrête brusquement à une valeur d'environ 0,18. Elle n'a pas eu le temps d'atteindre l'asymptote (proche de 0).
- Poids insuffisants : Les poids augmentent dans la bonne direction mais leur valeurs finales sont trop faibles. De plus, on observe des oscillations régulières. Cela vient que le réseau corrige ses poids mais vu qu'on a un nombre faible d'itérations, on les voit, alors qu'avec un grand nombre d'itérations, elles sont noyées dans la tendance globale.

Conclusion

Cette partie a montré que :

- on doit avoir un nombre d'itérations minimum, afin de permettre au réseau neuronal, d'avoir un apprentissage correcte (que l'erreur puisse converger à 0)
- Pour la vitesse (`coeff`), il ne doit pas être trop lent (le réseau a une converge interminable), ni trop rapide (le réseau est instable et "brutale").

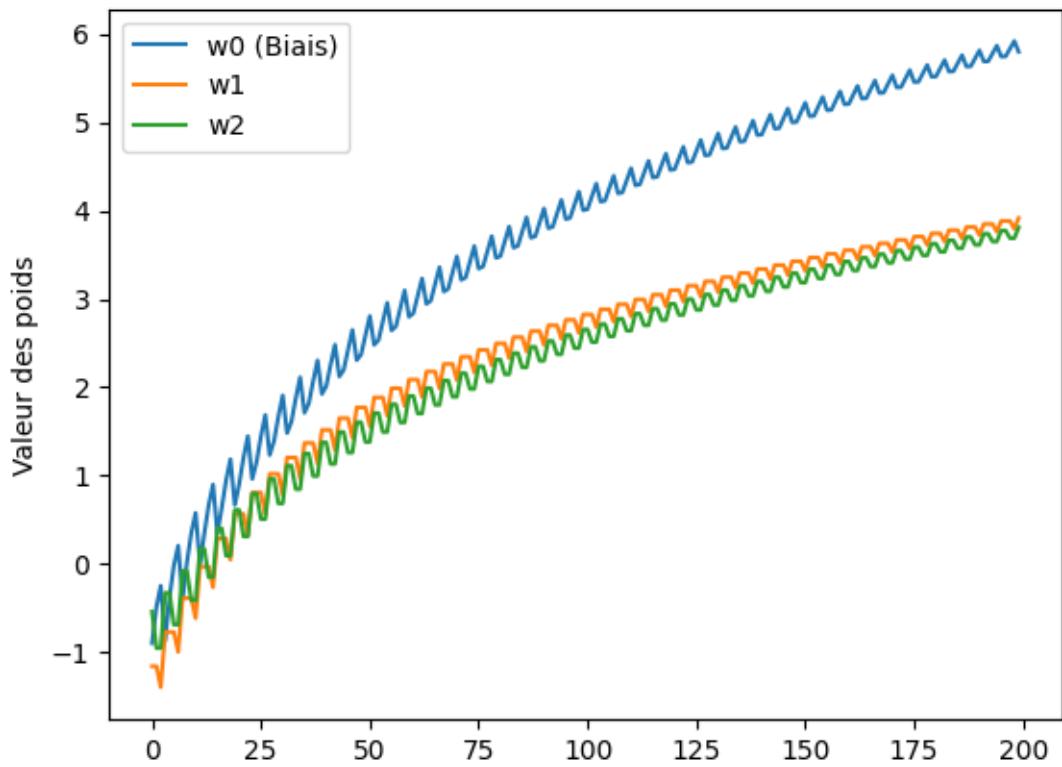


Figure 7: Evolution des poids en fonction du nombre d'itération avec le paramètre `iterations` à 50

Modification pour une porte OR

Pour que le perceptron s'entraîne sur une porte OR au lieu d'une porte AND, il faut changer `desired_out`.

Pour qu'il sorte une porte OR, il faut avoir `desired = np.array([0, 1, 1, 1])`

Ainsi, on obtient ces valeurs de poids :

- $w_0 = 4,92$
- $w_1 = 10,77$
- $w_2 = 10,77$

Voici le graphique de l'évolution de l'erreur :

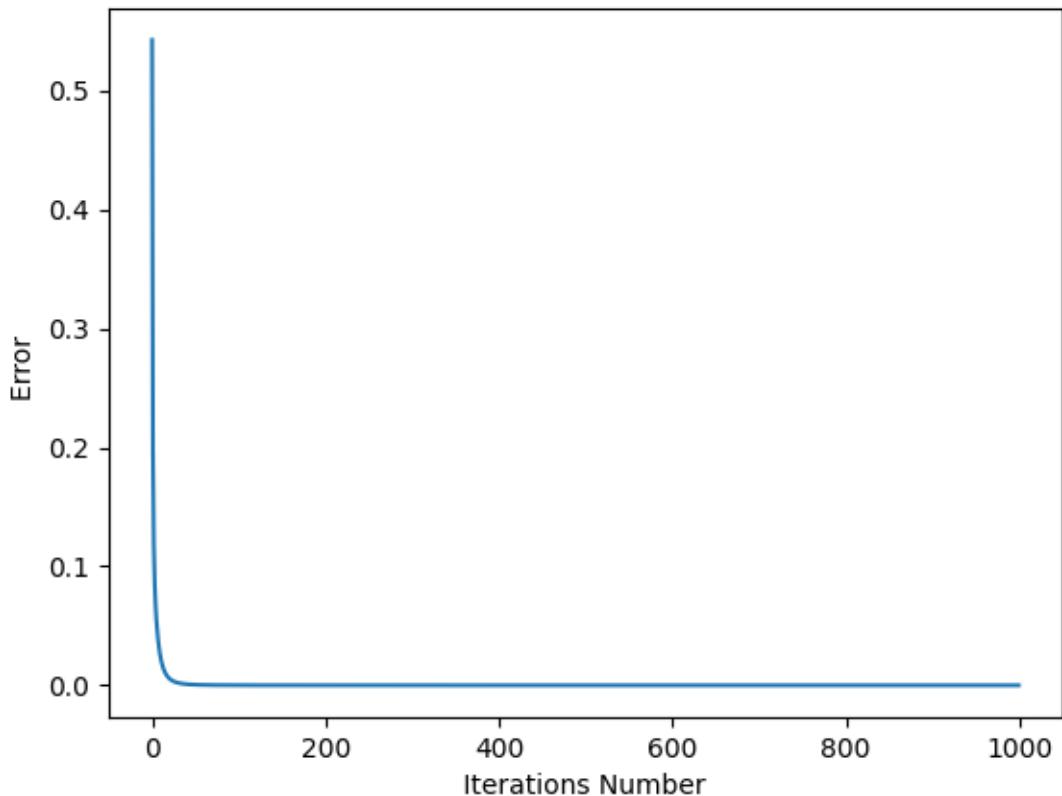


Figure 8: Evolution de l'erreur en fonction du nombre d'itérations avec la porte OR

Et le graphique de l'évolution des poids :

On voit que :

- l'erreur a convergé vers 0 très rapidement, prouvant la facilité du problème pour le perceptron.
- la hiérarchie des poids ont été inversé : contrairement à la porte AND, on observe que les courbes orange (w_1) et (w_2) passent au-dessus de la courbe w_0 . La symétrie entre w_1 et w_2 est conservé.

w_1 et w_2 sont supérieurs au biais (w_0) car la porte OR est égale à déjà qu'une des deux entrées est égale à 1 (ou les deux).

Ainsi, on a deux cas :

- $(0, 0)$: on a $-4,92 + 0 + 0 = -4,92$, le calcul est négatif ainsi la sortie est égale à 0

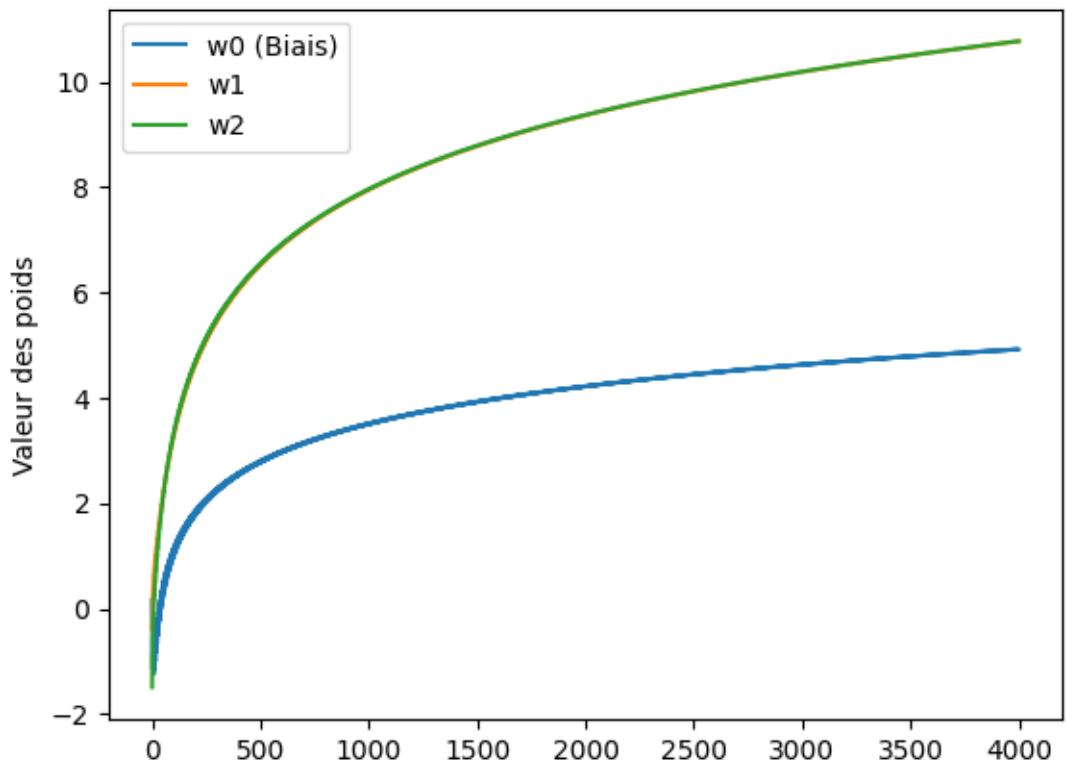


Figure 9: Evolution des poids en fonction du nombre d'iérations avec la porte OR

- $(0, 1), (1, 0)$: on a $-4,92 + 10,77 = 5,84$, le calcul est positif ainsi la sortie est égale à 1
- $(1, 1)$: on a $-4,92 + 10,77 + 10,77 = 16,62$, le calcul est positif ainsi la sortie est égale à 1.

Donc, on a un biais bas pour que, dès qu'une entrée soit égale à 1, la valeur de cette entrée dépasse le biais et le résultat devient positif.

Modification pour une porte XOR

Pour que le perceptron s'entraîne sur une porte XOR au lieu d'une porte OR, il faut changer `desired_out`.

Pour qu'il sorte une porte XOR, il faut avoir `desired = np.array([0, 1, 1, 0])`

On trouve, ainsi, ces valeurs :

- $w_0 = -0,4$
- $w_1 = -0,8$
- $w_2 = -0,4$

Voici le graphique de l'évolution de l'erreur :

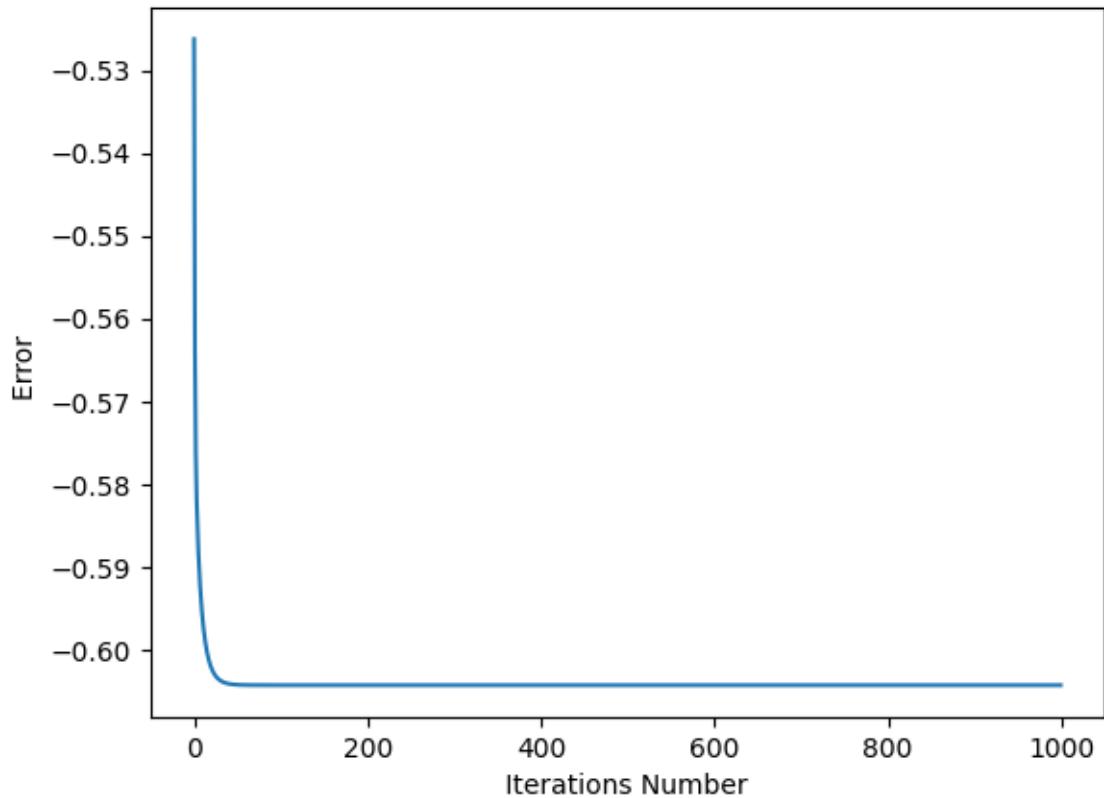


Figure 10: Evolution de l'erreur en fonction du nombre d'itérations avec la porte XOR

Et le graphique de l'évolution des poids :

Avec ces graphiques et valeurs, on peut tirer ces observations :

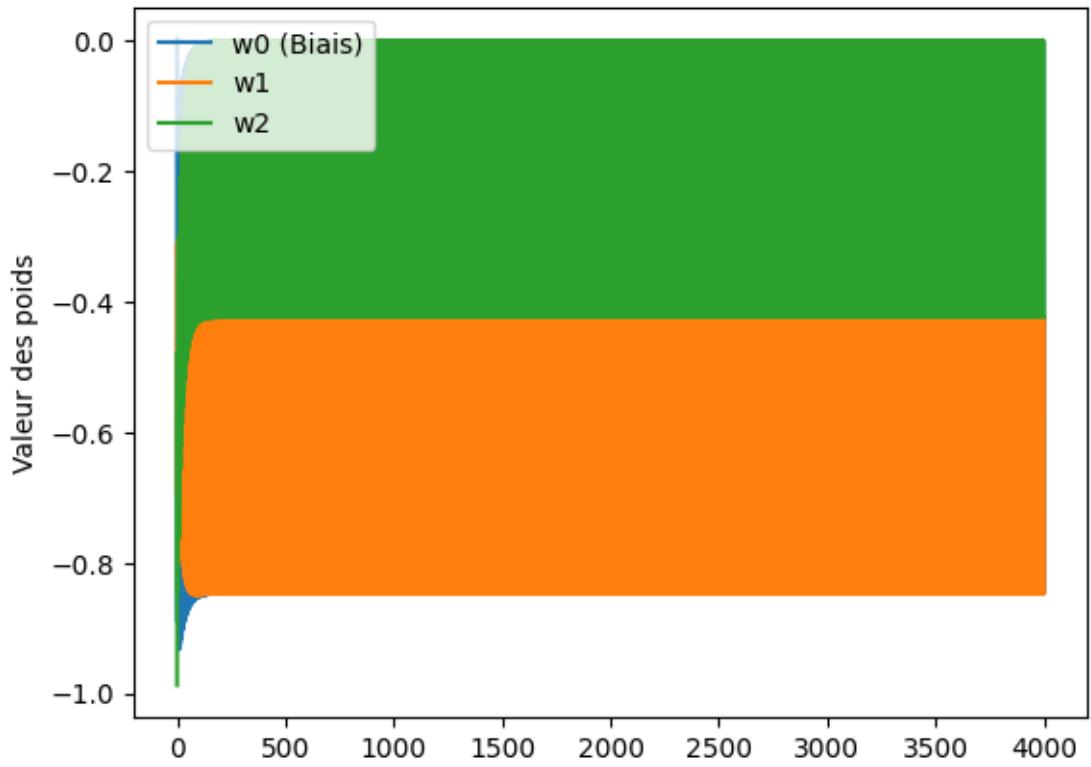


Figure 11: Evolution des poids en fonction du nombre d'itérations avec la porte XOR

- Stagnation de l'erreur : Contrairement aux cas des portes AND et OR, la courbe de l'erreur ne descend jamais vers 0. Elle chute initialement et se bloque à une valeur élevée (environ 0,6). Cela signifie que le réseau se trompe grossièrement sur au moins 2 des 4 exemples en permanence.
- Instabilité chronique des poids : Au lieu de lignes fines et stables, on observe des “bandes” épaisses de couleur (surtout pour w_1 en orange et le biais w_0 en bleu). Cela indique une oscillation permanente : à chaque itération, le perceptron ajuste ses poids pour corriger une erreur sur un exemple (ex: (0,1)), mais cette correction crée immédiatement une nouvelle erreur sur un autre exemple (ex: (1,1)). Il tourne en rond sans jamais trouver d'équilibre.
- Valeurs finales incohérentes : les poids finaux ($w_0 \approx -0,4$, $w_1 \approx 0,8$, $w_2 \approx -0,4$) ne permettent aucune logique claire. Par exemple, si on prend ces poids et l'entrée (0,0), on trouve $y = (-1 \times (-0,4)) + 0 + 0 = 0,4$, le calcul est positif donc la sortie vaut 1 (alors que pour le XOR, le (0,0) donne 1).

Cette porte XOR montre les limites majeures du perceptron mono-couche.

Pour séparer les sorties valant 0 de celles valant 1, il est géométriquement impossible de tracer une seule ligne droite.

Les points (0,0) et (1,1) sont situés dans une diagonale.

Les points (0,1) et (1,0) sont sur l'autre diagonale.

L'algorithme cherche une solution qui n'existe pas dans son espace de recherche. Pour résoudre ce problème, il est impératif d'utiliser un perceptron multi-couches (MLP) qui pourra tracer deux lignes (ou une courbe pour isoler les classes).

Conclusion

Ce TD a permis d'appréhender les mécanismes fondamentaux de l'apprentissage supervisé à travers l'étude d'un perceptron mono-couche.

Les expérimentations menées ont mis en évidence trois points majeurs :

L'efficacité sur les problèmes linéaires : Le perceptron a parfaitement réussi à modéliser les fonctions logiques AND et OR. L'analyse des poids finaux a permis de comprendre comment le neurone “configure” sa frontière de décision (une droite) en ajustant le rapport de force entre le biais (seuil d'activation) et les poids des entrées.

L'importance cruciale des hyperparamètres : Les tests ont démontré que la réussite de l'apprentissage dépend d'un équilibre délicat. Un taux d'apprentissage (`coeff`) trop faible rend la convergence infiniment lente, tandis qu'un taux trop élevé provoque une instabilité (oscillations) empêchant la minimisation de l'erreur.

La limite structurelle du neurone unique : L'échec systématique sur la porte XOR constitue le résultat le plus significatif. Il illustre la limitation théorique du perceptron simple : son incapacité à résoudre des problèmes non linéairement séparables. L'algorithme tourne indéfiniment sans trouver de solution car aucune droite ne peut séparer les classes du XOR.

En conclusion, si le perceptron est un classifieur linéaire efficace, la résolution de problèmes complexes (comme le XOR) nécessite de passer à des architectures plus évoluées, telles que le perceptron multi-couches (MLP), capable de générer des frontières de décision non linéaires grâce à l'ajout de couches cachées.

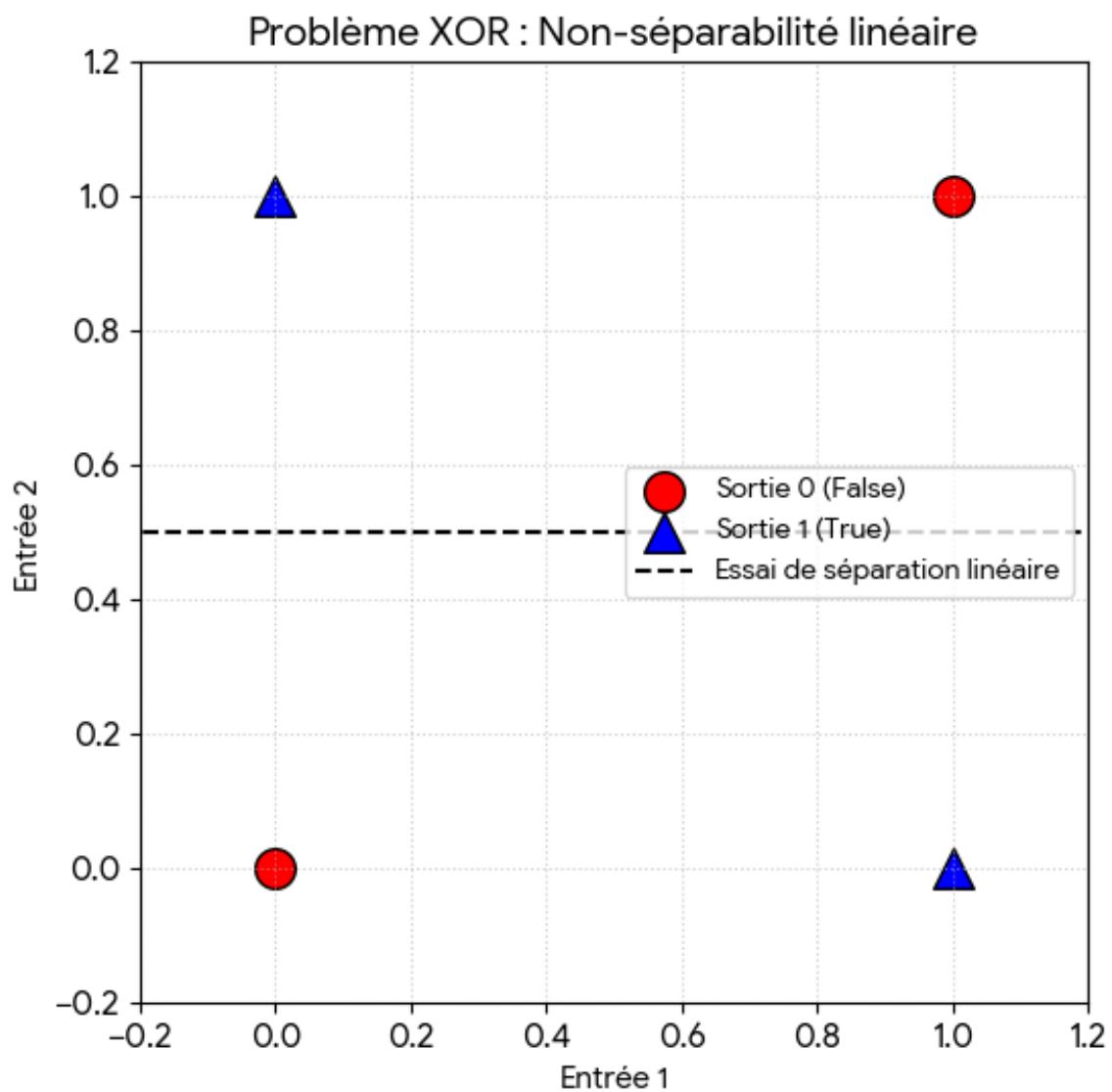


Figure 12: Représentation graphique du problème avec les solutions

TD 2 - GAN

Introduction

Dans ce TD, on va étudier deux implémentations de GAN (*Generative Adversarial Network*). Un GAN est une classe d'algorithme d'apprentissage non-supervisé, utilisant deux réseaux neuronaux placés en compétition, un jeu. Le premier est appelé “générateur” : celui-ci génère un échantillon (ex: image, position de points,...). Le deuxième est appelé “discriminateur” : celui-ci essaie de déterminer si les points du générateur sont vrais ou faux. Le but du générateur est de tromper le discriminateur. Le jeu doit être dit à somme nulle. Ca veut dire que si un joueur gagne, cela constitue la partie pour l'autre joueur.

GAN Lab

GAN Lab permet de simuler un réseau GAN et de modifier les hyperparamètres.

On choisit la distribution des points réels, puis on peut lancer le modèle. On retrouve nos deux systèmes neuronaux :

- Le générateur
- Le discriminateur

Mais pas que, on observe le bruit et les liens entre ces trois éléments.

Ainsi, le générateur s'appuie du bruit pour générer ses “faux” points. Le discriminateur est constamment relié aux bons points, afin d'apprendre leurs positions et de comprarer avec les faux points du générateur. Le discriminateur sort une liste de points qu'il considère comme faux. Avec cette liste de points, on calcule la perte du générateur (c'est le nombre de points faux détectés par le discriminateur divisés par le nombre de points total) et celle du discriminateur (le nombre de points faux non détectés divisés par le nombre de points total). Avec ces pertes, on peut calculer les gradients et alimenter les deux systèmes neuronaux afin d'améliorer, pour le générateur, la non-détection des faux points, et pour le discriminateur, la détection des faux points.

Avec à l'interface, on peut identifier plusieurs hyperparamètres :

- la génération du bruit (choix entre le nombre de dimension, et la formule mathématique)
- le nombre de couches et de neurones du générateur
- le nombre de couches et de neurones du discriminateur
- le nombre de mise à jour par entraînement

Analyse des hyper-paramètres

Maintenant, voyons comment la variation des hyper-paramètres résulte sur le modèle GAN.

Génération du bruit

La génération de bruit est le point de départ du générateur pour créer une image.

Ce hyper-paramètre se constitue en deux sous-paramètres :

- la dimension
- la distribution mathématique

La variation de dimension permet de montrer que ce paramètre limite la variété des images générées. Ainsi, le générateur ne va pas pouvoir placer des points qui sont éloignés des autres, comme on voit sur l'image :

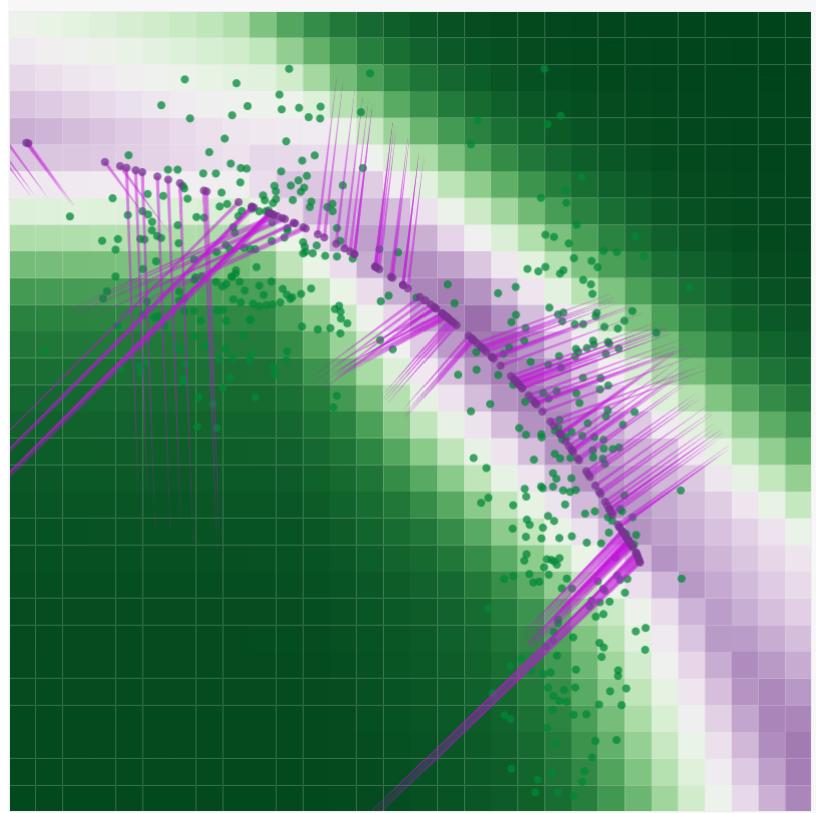


Figure 13: Génération avec une distribution gaussienne 1D

Sur l'image, on a mis la génération du bruit sur une dimension utilisant une distribution gaussienne. Les points générés (et découverts) ne suivent qu'une ligne (alors que les points réelles sont sur une distribution 2D).

Le deuxième sous-paramètre, la distribution mathématique. Dans GANlab, on a deux types :

- uniforme
- gaussienne

La distribution uniforme fait que les points de départ ont la même probabilité d'être choisis. La distribution gaussienne fait en sorte que les points de départ sont concentrés autour du centre, avec moins de points sur les bords.

Couches et neurones du Générateur

Dans GANlab, on peut paramétriser le générateur sur deux paramètres :

- le nombre de couches
- le nombre de neurones

Si on réduit le nombre de couches et de neurones, le générateur est “simple” (faible capacité). Il a du mal à apprendre des formes complexes. Les points orange forment probablement un “blob” simple ou une ligne.

Au contraire, si on augmente le nombre de couches et de neurones, le générateur est “intelligent”. Il apprend à faire des formes très complexes à partir du bruit. Sauf que si il est trop puissant, il peut devenir long à entraîner ou instable.

Couches et neurones du Discriminateur

Dans GANlab, on peut paramétriser le discriminateur sur deux paramètres :

- le nombre de couches
- le nombre de neurones

Si on réduit le nombre de couches et de neurones, le discriminateur est “simple” (faible capacité). Il va être facile à tromper. Le générateur va “gagner” rapidement mais le résultat va être mauvais car le “policier” n'est pas assez compétent pour le forcer à s'améliorer.

Au contraire, si on augmente le nombre de couches et de neurones, le discriminateur est “intelligent”. Il apprend très bien à définir la frontière entre les vrais points et les faux. Sauf que si le discriminateur est beaucoup trop fort que le générateur, il devient “parfait” trop vite. Le générateur, face à un adversaire parfait, ne trouve plus aucune piste à s'améliorer (ses gradients s'annulent) et il s'effondre. Il peut aussi trouver une seule faille chez le discriminateur et ne produire que cette sortie.

Nombre de mises à jour par entraînement

Ce nombre de mises à jour par entraînement est le nombre d'entraînement de chaque réseau (générateur et discriminateur) à chaque étape. Le nombre de mises à jour du générateur et du discriminateur forme le rapport d'entraînement.

Si on prend un rapport de 1:1 (discriminateur:générateur), le discriminateur et le générateur apprend au même rythme. Sauf que c'est instable, car si un des réseaux va prendre de l'avance, l'autre ne pourra pas suivre.

Si on prend un rapport 5:1, le discriminateur va toujours être un peu meilleur que le générateur. Cela permet au générateur d'avoir des indications claires et précises au générateur sur la manière de s'améliorer. Si le discriminateur est trop faible, ses conseils sont mauvais.

StyleGAN 3

Dans cette partie, on va étudier StyleGAN3. Ce modèle est la troisième itération de ce modèle de réseau de neurones génératifs (dit GAN) développée par les chercheurs de NVIDIA.

La principale innovation de cette itération est l'architecture “Alias-Free”. Cette architecture permet à StyleGAN3 de ne plus avoir d’“aliasing” (crénelage), ce qui permet :

- supprimer le “texture sticking” : l'ancienne version, quand elle animait une visage, certains détails (comme les cheveux, barbe,...) semblaient “collés” à l'écran plutôt de suivre le mouvement de la tête. Avec StyleGAN3, les textures suivent parfaitement la géométrie de l'objet.
- une vision “signal” et non “pixel” : l'intelligence artificielle ne traite plus l'image simplement comme une grille de pixels fixes, mais comme un signal continu. Cela permet des transformations géométriques (rotations, translation) beaucoup plus naturelles

Instanciation du modèle

Pour mettre en place l'environnement StyleGAN3, nous n'avons pas installé les dépendances directement sur la machine locale pour éviter les conflits de versions (notamment liés aux pilotes CUDA et à PyTorch).

Nous avons opté d'utiliser le conteneur Docker, en se basant sur le Dockerfile fourni dans le dépôt officiel de NVIDIA.

Les étapes réalisées sont les suivantes :

1. Construction de l'image Docker à partir du Dockerfile
2. Lancement du conteneur avec l'accès aux GPU
3. Téléchargement d'un modèle pré-entraîné (fichier `.pkl`). Nous avons utilisé un modèle entraîné sur le jeu de données FFHQ (Flickr-Faces-HQ), capables de générer des visages humains en haute résolution (1024×1024).

Réalisation de différentes inférences

Une fois, le conteneur actif, nous avons utilisé le script de génération (`gen_images.py`) pour créer des images synthétiques.

Le principe repose sur l'utilisation d'une graine aléatoire (seed). Cette graine (un nombre entier) sert de point de départ pour générer le vecteur latent. L'intérêt est que pour une même graine et un même modèle, l'image générée sera toujours identique, ce qui assure la reproductibilité.

Et voici trois exemples d'images générées avec des graines différentes.

Inférence 1 Dans cette inférence, on a utilisé pour graine 2.

Le modèle génère un visage masculin aux cheveux courts foncés. On remarque avec une texture de peau très détaillée, avec un aspect légèrement brillant (reflets ou transpiration), sur un fond abstrait aux tons rouges. Le regard est dirigé vers le coté.

Inférence 2 Dans cette inférence, on a utilisé pour graine 674.

Il s'agit d'un visage féminin aux cheveux clairs, pris en contr-plongée avec un fond bleu ciel lumineux. L'expression est dynamique (bouche ouverte, dents visibles), ce qui démontre la capacité du modèle à gérer des géométries faciales complexes et non statistiques.

Inférence 3 Dans cette inférence, on a utilisé pour graine 1024.

Le modèle produit ici le portrait d'une femme aux cheveux longs bruns avec des mèches plus claires. L'éclairage est doux et l'arrière-plan est flou, aux teintes jaunes et vertes, rappelant un environnement naturel ou automnal.

Analyse des résultats Les images obtenus démontrent la puissance de l'architecture StyleGAN3. Contrairement aux premiers GANs qui produisaient souvent des artefacts visuels ou des textures floues ("texture sticking"), les visages générés ici possèdent un niveau de détails (cheveux, paupières, éclairage) indiscernable d'une photo réelle. L'architecture "Alias-Free" permet une cohérence géométrique parfaite des textures.

Conclusion

Ce TD nous a permis d'aborder les réseaux antagonistes génératifs (GAN) sous deux angles complémentaires :

1. L'approche théorique et visuelle avec GAN Lab : Nous avons pu observer la dynamique de compétition entre le générateur (qui crée les faux) et le discriminateur (qui les détecte). Nous avons constaté l'importance cruciale de l'équilibre des hyperparamètres (taux d'apprentissage, complexité des réseaux) : si l'un des deux réseaux devient trop fort trop vite, l'apprentissage s'effondre.
2. L'approche pratique et "état de l'art" avec StyleGAN3: En utilisant Docker pour instancier un modèle complexe, nous avons manipulé un outil de recherche avancé. Cela a illustré comment les concepts vus dans GAN Lab (bruit, couches, distribution), une fois passés à l'échelle supérieure avec des architectures optimisées, permettent de générer des données synthétiques d'un réalisme saisissant.



Figure 14: Image générée par StyleGAN3 avec la graine 2

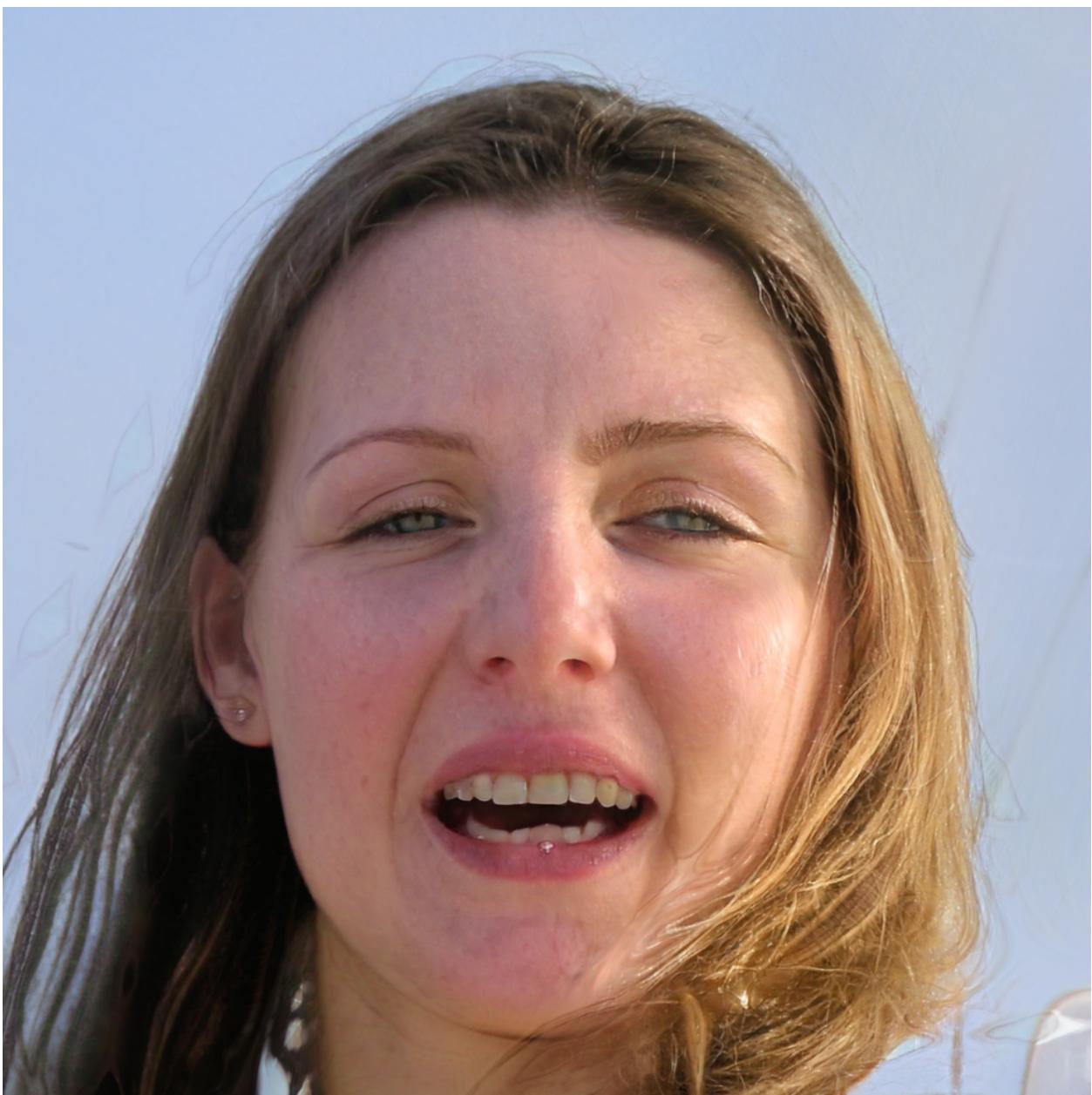


Figure 15: Image générée par StyleGAN3 avec la graine 674



Figure 16: Image générée par StyleGAN3 avec la graine 1024

TD 3 - Transformers

Introduction

L'objectif de ce travail dirigé est de se familiariser avec l'utilisation des modèles transformeurs pour le traitement du langage naturel (NLP). Nous avons utilisé la bibliothèque Python d'Hugging Face pour entraîner deux types de modèles de langage : le *Causal Language Modeling* (CLM) et le *Masked Language Modeling* (MLM).

Préparation de l'environnement et des données

Installation et authentification

Nous avons d'abord installé les bibliothèques nécessaires (`datasets` et `transformers`) et nous nous sommes connectés au Hub d'Hugging Face via `notebook_login` pour pouvoir sauvegarder nos modèles.

Dataset

Le jeu de données utilisé pour ce TP est Wikitest2. Celui-ci contient des extraits d'articles Wikipédia, jugés de qualité.

Voici un exemple de donnée brute issue du dataset :

“The game’s battle system, the BlitZ system, is carried over directly from Valkyria Chronicles...”

Causal Language Model

Principe

Ce type de modèle doit prédire le prochain token d'une phrase en se basant uniquement sur les tokens précédents (le masque d'attention empêche de voir le futur). C'est l'architecture utilisée par les modèles GPT.

Configuration

Pour cette partie, notre objectif est de créer un modèle du type GPT2. Pour créer les “tokens” (morceau de texte contenant un contexte syntaxique), on va le modèle `sgugger/gpt2-like-tokenizer`. De plus, la taille des blocs créés est de 128 tokens.

Entraînement

Avant, dans le notebook, on a créé les datasets d'entraînement et de validation. Leur taille est de 37000 exemplaire pour l'ensemble de données d'entraînement et 3700 pour celui de validation.

L'entraînement a été réalisée sur trois époques (un époque correspond au passage complet du jeu de données d'entraînement).

On a, ainsi, ces valeurs de pertes :

| Epoque | Perte d'entraînement | Perte de validation |
|--------|----------------------|---------------------|
| 1 | 6,507300 | 6,426713 |
| 2 | 6,146800 | 6,156285 |
| 3 | 5,968300 | 6,068984 |

Puis, on calcule la perplexité. La perplexité est le nombre de choix différents entre lesquels le modèle hésite en moyenne.

Son calcul est : $\text{Perplexité} = \exp^{perte}$

On trouve ainsi une perplexité de 432,24. Donc, le modèle, pour le choix d'un mot, hésite entre 432 mots différents.

Cette valeur est élevée, mais elle est expliquée par le fait qu'on a entraîné le modèle sur un petit ensemble de données et sur un nombre réduit d'époques pour les besoins de la démonstration. Le modèle a, néanmoins, réussi à réduire sa fonction de perte de manière constante.

Passons au MLM

Masked Language Modeling (MLM)

Principe

Ce type de modèle doit prédire des tokens, qui sont masqués aléatoirement dans l'entrée (remplacés par [MASK]). Contrairement au CLM, le modèle a accès au contexte situé à gauche et à droite du token à trouver.

Configuration

Pour cette partie, notre objectif est de créer un modèle du type BERT. Pour créer les "tokens", on va utiliser le modèle `sgugger/bert-like-tokenizer`. De plus, la taille des blocs créés est de 128 tokens.

L'élément en plus que le CLM est l'exécution d'un pré-traitement. La classe `DataCollatorForLanguageModeling` va masquer, dynamiquement, 15% des tokens (via le paramètre `mlm_probability`) à chaque époque.

Entraînement

Les ensembles de données sont les mêmes que pour le CLM, comme pour le nombre d'époques.

On a, ainsi, ces valeurs de pertes :

| Epoque | Perte d'entraînement | Perte de validation |
|--------|----------------------|---------------------|
| 1 | 7,098900 | 7,056674 |
| 2 | 6,909300 | 6,894692 |
| 3 | 6,859700 | 6,878888 |

Ainsi, on calcule la perplexité, et on trouve 963,88. Donc, le modèle hésite entre 963 mots pour choisir le token.

Analyse

La perplexité obtenue ici (963,88) est supérieure à celle du modèle GPT-2. Bien que la tâche MLM soit théoriquement plus simple car elle bénéficie de plus contexte, le résultat montre que le modèle BERT initialisé aléatoirement a eu du mal à converger rapidement avec si peu de données. Comme pour le CLM, un entraînement plus long sur un corpus plus vaste serait nécessaire pour obtenir des performances utilisables.

Conclusion

Ce TD a permis de mettre en oeuvre deux architectures majeures de l'état de l'art : GPT-2 (autorégressif) et BERT (auto-encodeur). Nous avons réussi à :

- Préparer et tokeniser le dataset Wikitest-2
- Configurer et entraîner sur le Hub Hugging Face (commit confirmé dans les logs).

Les valeurs de perplexité élevées (432 et 963) soulignent l'importance de la quantité de données et du temps de calcul (nombre d'époques) pour l'entraînement efficace des Large Language Models (LLMs).