Compte rendu : mini projet "classifier"

CROS Ivan, L3 Info UTLN

21 avril 2024

Introduction

Dans ce mini-projet, nous nous attelons à créer un classifieur empirique basé sur la théorie de Shannon, en utilisant la cross-entropie et la divergence de Kullback-Leibler. Ce classifieur sera capable de reconnaître des formes dans des images bruitées générées par notre programme. L'objectif global est de mieux comprendre l'utilisation de la théorie fondamentale et empirique de Shannon dans l'apprentissage automatique. **Je vous invite à lire directement la conclusion puis les étapes de réalisation du projet une à une pour une mieux comprendre le sujet et saisir l'importance de ce qu'il en retourne.**

Table des matières

1 Première partie : modélisation des classes et évaluation de l'information partagée.

1.1 Objectifs

Nous allons générer 2000 fichiers CSV, chacun une classe d'images parmi 6 classes représentant différentes forment aléatoires : coins, diagonales, antidiagonales, centres, lignes verticales ou horizontales.

Pour déterminer la classe de chaque image, nous calculons la moyenne des éléments a_{ij} et utilisons cette valeur pour définir le modèle ou la fonction de densité de probabilité (pdf_j = probability density function). Cette méthode permet de représenter les images en termes de probabilités plutôt que de masses, favorisant ainsi une approche plus probabiliste. En utilisant la fonction de densité de probabilité, nous allons ensuite déterminer les images appartenant à chaque classe en utilisant l'entropie croisée. Ce processus, bien que synthétique, demeure efficace même pour des images réelles.

Le calcul de classe par moyenne des a_{ij} : $modèle_j = \sum_i \frac{A_{ij}}{|A_{ij}|}$.

Donc on divise par le cardinal des A_ij pour des images avec des pixels entre 0 et 1, cela donne : $pdf_j = \frac{bary_j}{\sum_{x_i} Bary_j}$

On va d'abord calculer la cross entropie (notée CE) entre un modèle et un autre. Donc on va avoir le même type d'information.

1.2 Présentation du programme

Notre mini-projet est implémenté en 3 fichiers : *shannon.py* pour les fonctions liés à la théorie fondamentale de Shannon, *randmat.py* pour la génération et la manipulation de matrices, *kit.py* pour l'articulation des différents modules et *main.py* pour le rendu de notre projet. Le programme va engendrer un dossier *img* pour les *cvs* représentant nos images non bruitées, un dossier *rendu* pour sauvegarder notre matrice $P.D.F$ en csv, et PNG pour sauver les rendus de matplotlib. Ainsi, on part sur une bonne base. Pour commencer :

— On créer 6 fonctions qui génèrent les classes d'images et l'une qui calcul l'entropie croisée. Pour chaque matrice, les valeurs nulles sont à 10^{-16} pour éviter les différents artefacts avec le logarithme dans nos calculs.

— On utilise matplotlib (imshow) pour afficher visuellement les matrices de pixels générées par nos fonctions. 1

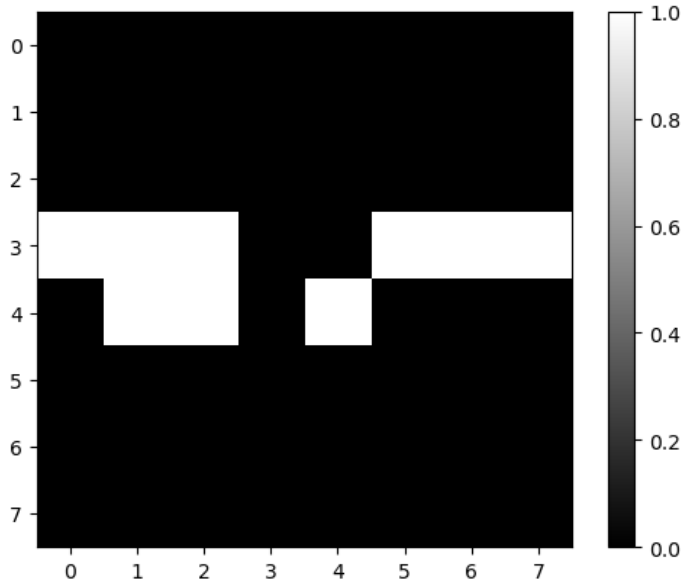

FIGURE 1 – Exemple d'une matrice de pixels en image avec matplotlib, pour générer pseudo-aléatoirement une ligne horizontale.

FIGURE 2 – Illustration du dossier 'data' généré

— On génère nos ensembles d'images 3 pour chaque classe 2 et à coder la fonction qui calcul la moyenne de classe.
— Il ne manque plus qu'à générer la matrice 4 mesurant l'entropie croisée entre chaque classe sur 1000 images d'entrainement que l'on séparera des 1000 autres pour nos différentes test.

1.3 Observations et déductions

Une entropie croisée élevée entre deux classes indiques des pixels communs dû à une similarité suggérant une classification plus confuse. Une entropie faible suggère une classification plus précise. La matrice des entropies fournit une vue d'ensemble de notre modèle de données.

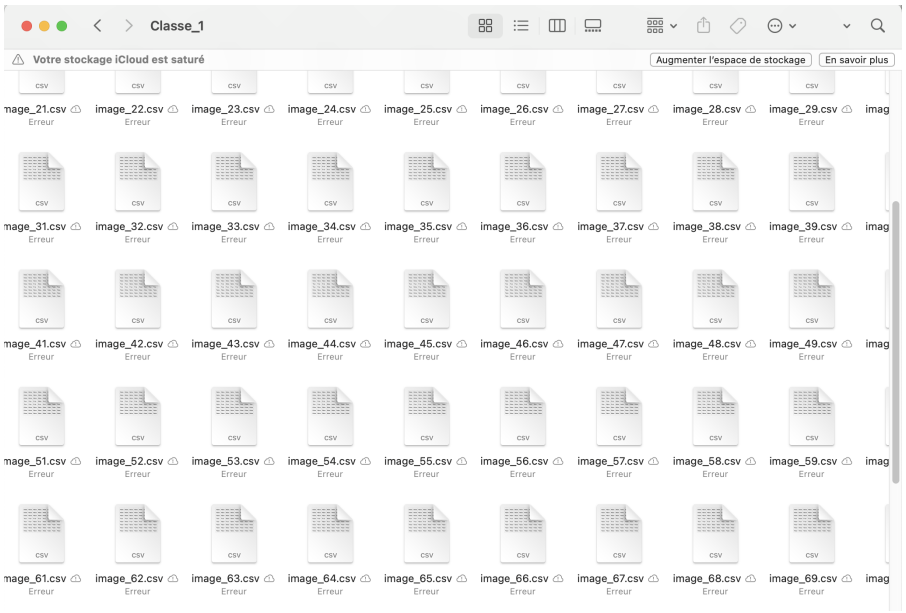

FIGURE 3 – Illustration des données csv générées pour la classe 1

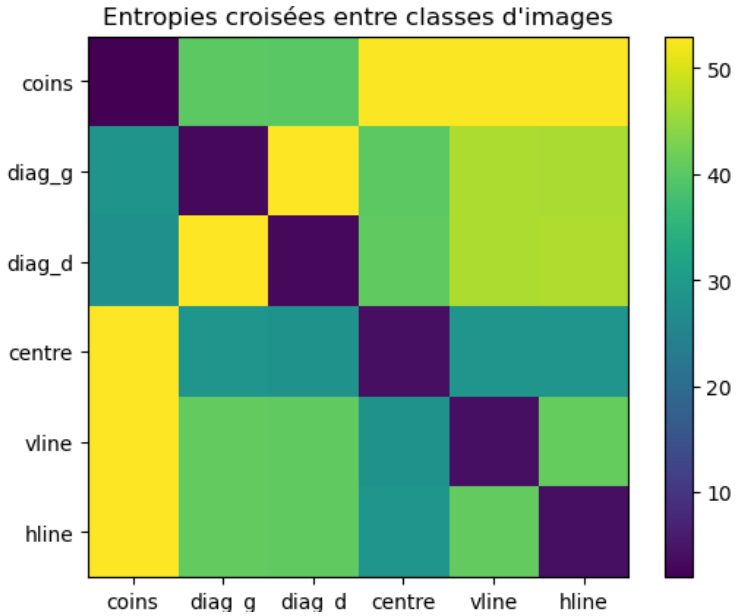

FIGURE 4 – Génération de la matrice représentant l'entropie croisée entre chaque classe pour un data train de 1000 images.

2 Deuxième partie : établissements de prédictions et traçage des courbes d'Accuracy

2.1 Objectifs

Pour déterminer la classe de chaque image, nous calculons la moyenne des éléments a_{ij} et utilisons cette valeur pour définir le modèle ou la fonction de densité de probabilité (PDF). Cette méthode permet de représenter les images en termes de probabilités plutôt que de masses, favorisant ainsi une approche plus probabiliste. En utilisant la fonction de densité de probabilité, nous allons ensuite déterminer les images appartenant à chaque classe en utilisant l'entropie croisée. Ce processus demeure efficace même pour des images réelles. Le calcul de classe par moyenne des a_{ij} : modèle $= \frac{\sum_{ij} P(A_{ij}) \cdot ij}{|A_{ij}|}$. Donc on divise par le cardinal des A_{ij} pour des images avec des pixels entre 0 et 1, cela donne : $PDF_j = \text{bary}_j \cdot \frac{\sum \text{Bary}_j}{|Bary_j|}$.

2.2 Observation

On veut améliorer la qualité de l'accuracy par l'échantillonnage. Si les données sont de mauvaise qualité, un classifieur produira invariablement des prédictions peu fiables. Pour obtenir des courbes plus lisses, il est souvent nécessaire de procéder à de nombreux échantillonnages, par exemple 1000 fois au lieu de 500. **Le bruit avec un signal dB près de 0 permet aucune prédiction fiable, et plus ce dernier augmente, plus nos prédictions se stabilisent et tendent vers une valeur.**

2.3 Résultats

On aurait espéré une courbe $diag_g$ et $diag_d$ logiquement supérieur aux autres maius ces dernière dont la $diag_d$ demeurent en dessous malgré mes ajustements. On obtient les accuracies suivantes :

[[0.0, 2.4, 15.5, 31.4, 50.7, 65.5, 76.9, 84.5, 87.8, 92.4, 93.5, 95.2, 94.8, 97.1, 96.8, 97.3, 98.2, 98.0, 98.1, 98.7, 98.7, 98.7, 98.7, 98.7, 98.7], [0.0, 0.9, 5.1, 21.8, 44.2, 62.6, 73.7, 81.5, 86.1, 89.0, 91.9, 94.8, 95.3, 95.6, 97.0, 96.6, 98.3, 98.7, 98.1, 99.3, 99.3, 99.3, 99.3, 99.3, 99.3], [0.0, 0.6, 6.0, 22.2, 43.3, 61.9, 74.1, 80.0, 83.6, 86.0, 88.1, 91.3, 91.0, 91.3, 92.1, 92.4, 93.7, 92.7, 92.5, 94.2, 94.2, 94.2, 94.2, 94.2, 94.2], [0.0, 0.0, 2.4, 9.7, 32.8, 47.8, 66.3, 77.9, 84.6, 87.6, 93.3, 94.5, 95.3, 96.0, 97.9, 97.6, 98.4, 98.6, 98.3, 99.0, 99.0, 99.0, 99.0, 99.0, 99.0], [0.0, 0.1, 2.6, 17.7, 42.3, 64.0, 78.5, 88.9, 92.8, 95.8, 98.3, 98.5, 99.6, 99.4, 99.4, 99.7, 99.9, 99.9, 99.9, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0], [0.0, 0.2, 3.3, 16.7, 45.5, 62.6, 78.4, 90.0, 94.3, 96.0, 97.7, 98.5, 98.8, 99.0, 99.8, 99.7, 100.0, 99.9, 99.9, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]]

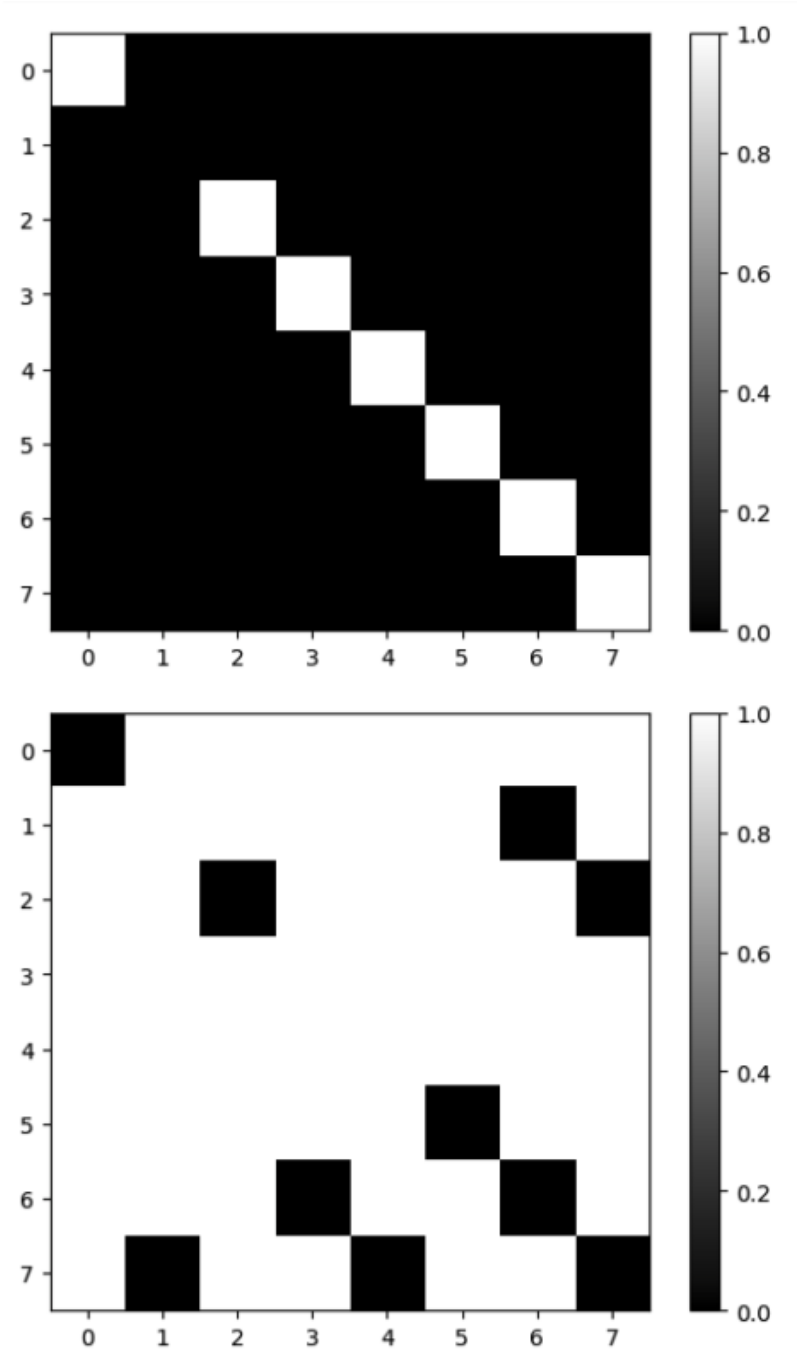

FIGURE 5 – Exemple d'une image non bruité (au-dessus) et de la même image bruité (en dessous) générées par notre programme.

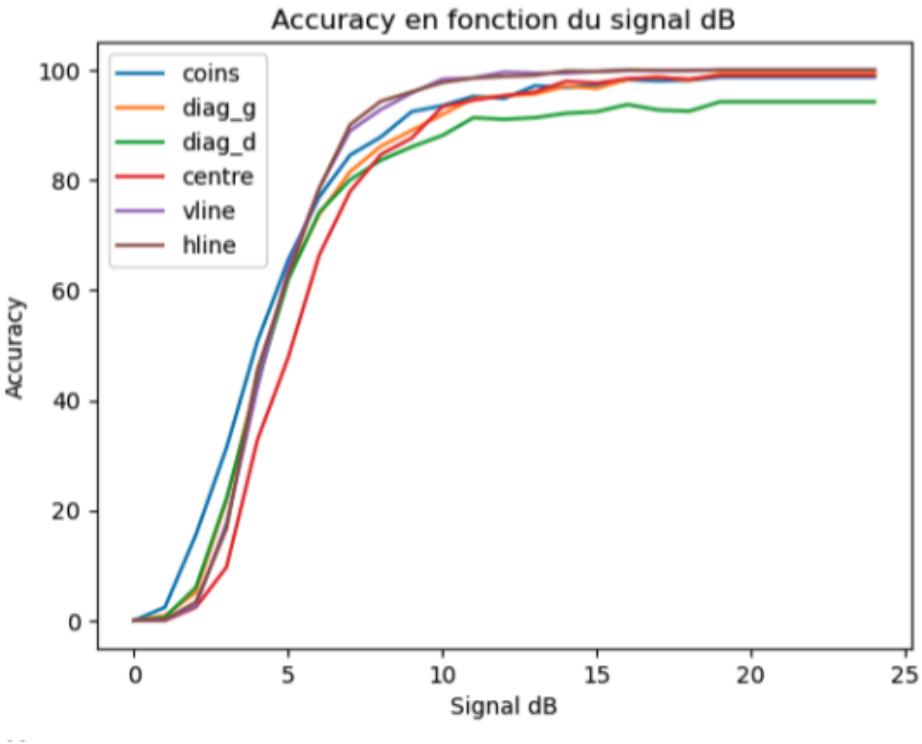

FIGURE 6 – Réalisation des courbes d'accuracy pour un bruitage jusqu'à 25 sur 1000 tirages des images 1001 à 2000 (data-test).

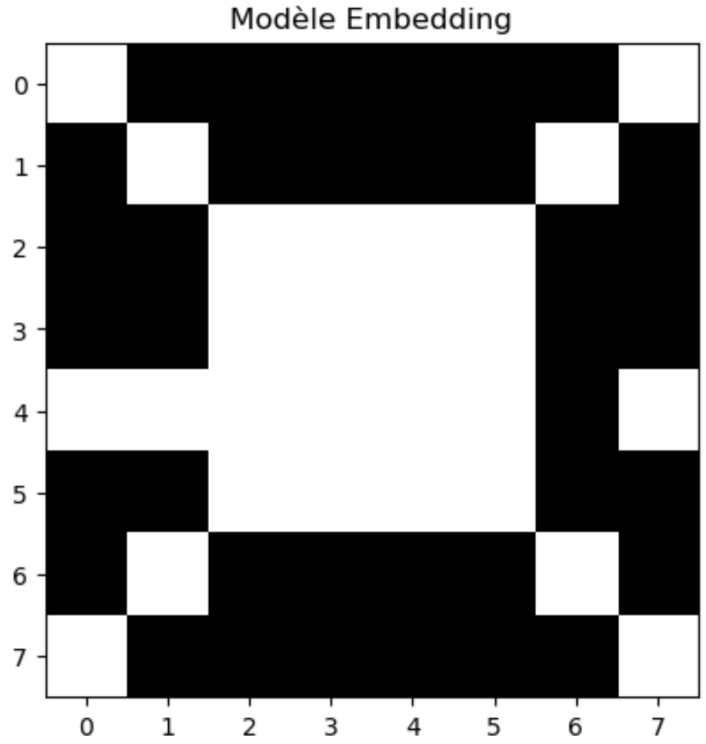

FIGURE 7 – Affichage des pixels les plus discriminants (en blanc) sur une image de 32 pixels.

3 Troisième et dernière partie : optimisation de nos résultats à l'aide de la divergence de Kullback-Leiber (divKL).

3.1 Objectifs

Pour chaque pixel de 1 à 53, nous allons calculer la $divKL(p_{jA}||p_{jB})$ pour tous les couples de classe, retenir la moyenne DIV(j), trier les 64 pixels par ordre décroissant et ne conserver que les 32 premiers pixels à conserver étant les plus discriminant à évaluer, ou les 32 derniers à retirer (ce que nous allons faire) étant ceux à remplacer par epsilon $:= 10^{-16}$ dans nos images avant de les bruiter pour optimiser notre classification.

3.2 Résultats

On obtient **??** : [[0.0, 1.8, 14.7, 31.9, 54.0, 66.3, 77.2, 82.6, 87.9, 89.8, 94.6, 94.4, 96.0, 95.9, 97.2, 96.6, 98.0, 97.7, 98.0, 98.7, 98.7, 98.7, 98.7, 98.7, 98.7], [0.0, 0.7, 6.6, 23.5, 43.9, 62.4, 72.6, 82.8, 86.8, 88.2, 91.2, 94.7, 94.2, 97.0, 97.2, 97.2, 98.2, 98.0, 97.9, 99.3, 99.3, 99.3, 99.3, 99.3, 99.3], [0.0, 0.9, 5.7, 24.1, 42.9, 62.3, 71.1, 79.5, 83.2, 85.4, 87.6, 88.8, 89.8, 90.8, 92.2, 91.8, 92.6, 93.4, 92.8, 94.2, 94.2, 94.2, 94.2, 94.2, 94.2], [0.0, 0.2, 2.5, 11.0, 32.6,

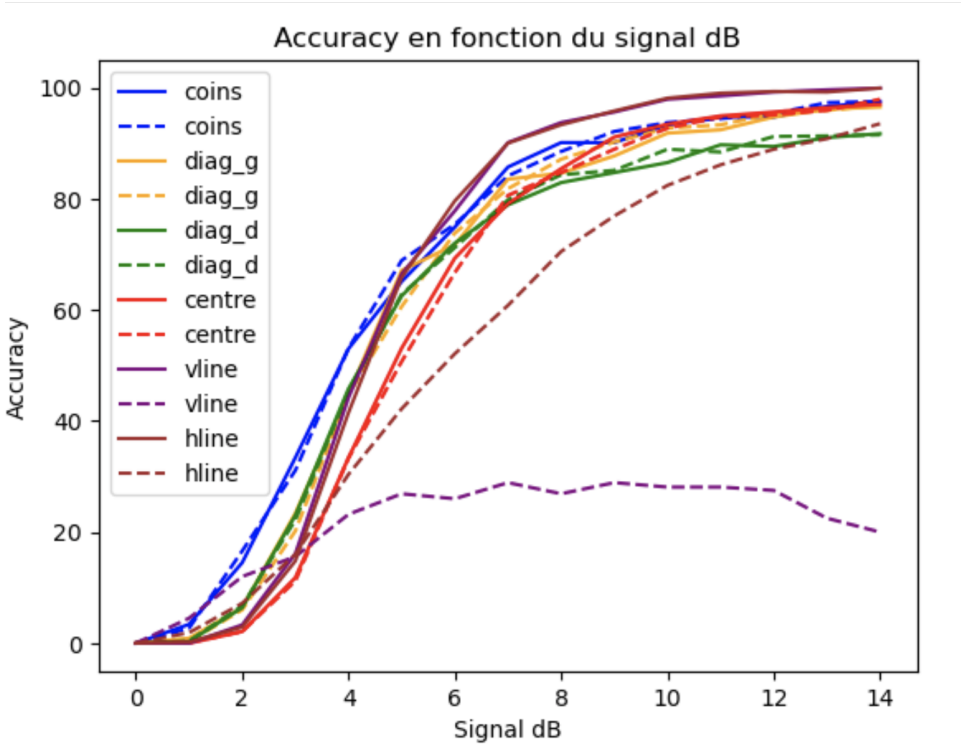

FIGURE 8 – Réalisation des courbes d'accuracy, en courbes pointillées pour un bruitage jusqu'à 15 sur 1000 tirages (data-test) avec optimisation sur pixels discriminants, en courbes pleines sans bruitage sur 1000 tirages toujours.

50.0, 64.9, 78.2, 86.1, 88.9, 92.6, 94.4, 95.2, 95.8, 97.4, 97.2, 98.4, 98.3, 98.3, 99.0, 99.0, 99.0, 99.0, 99.0, 99.0]], [0.0, 0.1, 3.0, 17.0, 43.9, 64.1, 80.2, 89.8, 92.7, 95.3, 98.1, 98.3, 98.6, 99.3, 100.0, 99.9, 99.9, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]], [0.0, 3.6, 10.2, 17.0, 23.3, 28.5, 30.0, 35.6, 32.5, 38.7, 39.9, 43.2, 41.7, 47.0, 44.9, 42.7, 47.7, 45.9, 45.2, 46.8, 46.8, 46.8, 46.8, 46.8, 46.8]]

Contre : [[0.0, 2.9, 16.2, 31.7, 52.2, 67.1, 76.7, 84.9, 89.2, 91.7, 94.1, 95.2, 96.1, 96.4, 96.5, 97.7, 98.1, 98.1, 97.7, 98.7, 98.7, 98.7, 98.7, 98.7, 98.7], [0.0, 0.4, 7.1, 23.6, 46.5, 61.5, 72.8, 79.9, 84.5, 90.4, 92.1, 93.1, 94.3, 95.7, 97.0, 96.4, 98.3, 98.3, 97.9, 99.3, 99.3, 99.3, 99.3, 99.3, 99.3], [0.0, 0.5, 6.2, 22.8, 45.0, 57.1, 73.3, 79.5, 85.9, 85.8, 89.0, 89.2, 90.7, 91.7, 91.9, 92.6, 92.8, 93.0, 92.8, 94.2, 94.2, 94.2, 94.2, 94.2, 94.2], [0.0, 0.0, 1.6, 12.2, 30.2, 50.3, 68.8, 79.4, 84.4, 88.5, 92.7, 94.9, 94.3, 96.0, 97.4, 97.9, 98.5, 98.3, 98.8, 99.0, 99.0, 99.0, 99.0, 99.0, 99.0], [0.0, 0.3, 2.5, 17.3, 44.3, 62.5, 80.5, 89.9, 93.1, 96.7, 98.4, 98.3, 99.2, 99.7, 99.8, 99.9, 100.0, 99.9, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0], [0.0, 0.0, 2.8, 17.6, 46.5, 63.8, 78.2, 90.0, 92.7, 95.6, 97.6, 98.6, 98.8, 99.7, 99.7, 99.7, 100.0, 99.9, 99.9, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]]

3.3 Déductions des observations

Donc après l'optimisation, on observe une sensible amélioration de l'exactitude des prédictions avec une augmentation du pourcentage d'exactitude de prédiction. Cela suggère que l'approche de sélection des pixels les plus discriminants pour la classification a amélioré les performances du modèle. **Toutefois, certaines classes ont légèrement diminué.** Cela peut être dû à **sur-ajustement** pour certaines classes dont la sélection des 32 pixels est plus pertinente mais introduit une perte d'information pour d'autres. Une analyse plus approfondie me permettrait d'améliorer l'implémentation et d'obtenir des performances plus stables.

4 Conclusions

4.1 Concernant l'aspect de nos courbes

Tout d'abord, l'observation de nos courbes révèle une forme sigmoïdale, soulignant **un phénomène de symétrie** et présentent une pente similaire, ce qui veut dire que ces courbes réagissent uniformément au bruit, démontrant **un impact homogène**. **La caractérisation de cette pente par rapport à 50% révèle que la moitié des pixels discriminants suffit à capturer une proportion significative** du pouvoir total de ma collection de donnée. **Plus on réduit la dimensions notre mon calcul de cross-entropie moyenne, mieux c'est, plus c'est optimal.**

Néanmoins, on ne retrouve pas les résultats que l'on s'imaginait mentalement au début de l'expérience. Notre perception mentale est biaisée par notre propre attention sélective : on est abusé l'importance de la surfréquence de certaines observation.

En outre, il est important de reconnaître que la perception des machines est analytique, tandis que celle des humains est influencée par des facteurs

subjectifs.

4.1.1 Concernant dKL et la cross-entropie

Nous résultats soulignent l'importance de l'échantillonnage sur les modèles qui sont à la fois symétriques et empiriques. Lorsque l'on utilise la divergence de Kullback-Leibler, nous mesurons l'entropie relative entre deux distributions en recherchant le maximum pour j, ce qui vise à trouver la plus grande séparation entre ces deux distributions.

En revanche, **une divergence avec une grande valeur ne fournit pas d'échelle de comparaison.** Néanmoins, il existe toujours un sous-espace de dimension dans lequel l'information est concentrée. C'est pourquoi nous réduisons le nombre de pixels les plus discriminants à moins de 32 pour optimiser nos prédictions lorsque l'on affiche **notre modèle embedding : il s'agit de l'espace latent.**

À la différence, la cross-entropie nous permet d'ordonner les éléments. Avec la théorie de l'information, on peut extraire l'espace latent par qualité d'entropie plutôt qu'un système de pondération par descente de gradient avec un réseau de neurones. **En réalité, c'est l'entropie qui minimise les coûts.**

Par ailleurs, on peut évoquer dans la même philosophie qu'il est préférable de récupérer le maximum de vraisemblance plutôt que d'utiliser le argmin(Mean Square Error) pour plus de précision.

La dKL et la cross-entropie sont deux outils précieux qui nous économise du temps et de l'énergie lorsque l'on sait s'en servir ! En tout cas, c'est ce que démontre ce mini-projet.

4.2 Concernant notre modèle

Pour répondre à notre introduction, ce modèle représente un réseau de neurones à 0 couche, sans avoir une fonction de coût définie puisque nous utilisons la cross-entropie, mais le principe reste similaire. En d'autres termes, notre rétro-propagation de l'erreur consiste à mesurer la cross-entropie.

Dans notre modèle, nous commençons d'abord avec des données non bruitées et symétriques, puis nous les rendons bruitées. Par conséquent, nous n'avons pas besoin de faire apprendre une représentation des données à notre modèle.

Ainsi très synthétiquement, notre réseau à une couche présente une solution divergente, ressemblant à un réseau de neurones avec une fonction logistique, agissant comme une fonction non linéaire pour transmettre les valeurs des pixels mais il est fondamentalement centré sur la cross-entropie.

$$\theta^* = \text{argmin}_\theta(f_\theta(CE(P\hat{D}F, PDF^*))) \tag{1}$$

4.3 Regard sur l'implémentation et améliorations possibles

La gestion des données

Notre collection de données pour le dataTrain et le dataTest pourrait rassembler l'ensemble de nos images enregistrer dans des fochiers cvs séparé en deux fichiers cvs

distincts pour moins de travail pour la machine et plus de rapidité. Par choix, j'ai généré autant qu'il y a de vrai images pour s'approcher au mieux du vrai.

La prédiction

Tout d'abord, il serait judicieux de retravailler la fonction $calculer_pixels_discriminants$ lié au modèle embedding pour un meilleur rendu. Je pense que celle-ci comporte une maladresse sur l'appel de $divergence_kl_pixel$. Ensuite, il serait plus judicieux de séparer l'affichage de plot pour les fonctions $plot_accuracy_vs_dB$ et $plot_accuracy_vs_dB_optimisee$ pour que notre dernière fonction $plot_accuracy$ soit lancé pus rapidement et sans encombrement.

Le style

Le code est organisé de façon à me permettre d'identifier mes différents retours et différentes valeurs, tapé à la volée de manière dynamique. Il pourrait être réparti de manière plus modulaire et moins chaotique.