Optimisation algorithmique et implémentation d'un classifieur fondé sur l'incompatibilité

Lebleu Alexandre, Jung Léonard
Encadrant : Chunyang Fan
Sorbonne Université

Mai 2025

Table des matières

1 Introduction **1**
1.1 Contexte 1
1.2 Motivation et objectifs du projet 1
1.3 Organisation du rapport 1

2 Fondements théoriques **1**
2.1 Situations, base de cas et mesures de similarité 1
2.2 Incompatibilité entre triplets 2
2.3 Prédiction par minimisation de l'incompatibilité 2
2.4 Algorithmes version naïve 2

3 Amélioration algorithmique **3**
3.1 Analyses mathématiques 3
3.2 Proposition Algorithme optimisé 7

4 Amélioration par parallélisation **7**
4.1 Introduction à pytorch 8
4.2 Passage du calcul en boucle au calcul parallèle 8

5 Expérimentations **9**
5.1 Implémentation 10
5.1.1 Fonctions communes aux versions en Python natif 10
5.1.2 Version naïve en Python 10
5.1.3 Version optimisée en Python 11
5.1.4 Version PyTorch 11
5.2 Résultats et expérimentations 11
5.2.1 Illustration de la prédiction 11
5.2.2 Comparaison versions Pythons 12
5.2.3 Version PyTorch 13

6 Limites et évolutions possibles **15**
6.1 Limites du calcul parallèle avec PyTorch 15
6.2 Améliorations possibles 15

7 Conclusion **16**

1 Introduction

1.1 Contexte

Complexity-based Analogical Transfer (CoAT) [**Badra**, **2020**] est une méthode de prédictions basées sur des cas (CBP) [**Janet L. Kolodner**, **1992**] qui relève du domaine du machine learning. Cette dernière vise à prédire le résultat d'un nouveau cas en s'appuyant sur des cas similaires déjà rencontrés. CoAT repose sur le transfert analogique [**Davis and Russell**, **1987**], un mécanisme cognitif selon lequel une solution à un nouveau problème peut être construite en s'inspirant d'un problème similaire déjà rencontré. En intelligence artificielle, ce principe est exploité en comparant des situations passées et en transférant la solution d'un cas source vers un cas cible, à condition que les deux soient suffisamment similaires. Ce transfert suppose une cohérence entre la similarité des situations (entrées) et celle de leurs solutions (sorties). Si deux cas sont proches dans l'espace des entrées, alors leurs sorties doivent aussi l'être, faute de quoi le raisonnement analogique échoue.

1.2 Motivation et objectifs du projet

La complexité algorithmique du classifieur CoAT constitue un obstacle à son application à grande échelle. En effet, lors du calcul de sa prédiction, de nombreuses comparaisons ont lieu entraînant un temps de calcul très élevé. L'auteur utilise notamment une **triple** boucle « for » pour effectuer les calculs, sans optimisation parallèle, ce qui laisse une marge d'amélioration pour l'algorithme. C'est cette double motivation, optimiser l'implémentation de CoAT et analyser son comportement sur un cas concret qui justifie le présent projet. Il vise à :

— la compréhension fine du fonctionnement de CoAT et de la mesure d'incompatibilité ;
— l'implémentation d'une version naïve en Python pour validation ;
— l'analyse de sa complexité et le développement d'une version optimisée ;
— le passage à une version parallèle avec PyTorch pour tirer parti du calcul vectoriel ;
— la comparaison expérimentale sur des jeux de données simulés ou réels.

1.3 Organisation du rapport

Dans la section suivante, nous détaillons les fondements mathématiques de la méthode CoAT et la définition formelle de l'incompatibilité. Nous présentons ensuite l'algorithme naïf et notre première version optimisée. La quatrième section est consacrée à l'implémentation parallèle en PyTorch. Enfin, nous analysons les performances obtenues, discutons des limites rencontrées et évoquons des perspectives d'amélioration.

2 Fondements théoriques

Dans cette section, nous introduisons les notations et définitions mathématiques qui serviront à formaliser le fonctionnement du classifieur CoAT.

2.1 Situations, base de cas et mesures de similarité

On note :

— $\mathcal{X}$: l'espace des entrées, c'est-à-dire l'ensemble des représentations possibles des situations ;
— $\mathcal{Y}$: l'espace des sorties, représentant les étiquettes ou résultats associés aux entrées.
— Une situation est définie comme une paire :

$$z_i = (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$$

où x_i est une entrée et y_i est la sortie associée.

— L'ensemble des cas constitue la base de cas, notée :

$$\mathcal{CB} = \{z_1, z_2, \ldots, z_n\}$$

avec n le nombre de cas observés.

Pour comparer les cas entre eux, on introduit deux fonctions de similarité :
— $\sigma_X : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, mesure de similarité entre les entrées ;
— $\sigma_Y : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, mesure de similarité entre les sorties.

Ces fonctions attribuent une valeur réelle indiquant à quel point deux éléments sont similaires. Plus la valeur est élevée, plus la similarité est forte, elles sont maximales si les entrées sont identiques.

2.2 Incompatibilité entre triplets

La méthode CoAT repose sur le principe que la similarité dans l'espace des entrées doit être cohérente avec celle dans l'espace des sorties.

Un triplet $(z_i, z_j, z_k) \in \mathcal{CB}^3$ est dit incompatible si :

$$\sigma_X(x_i, x_j) \geq \sigma_X(x_i, x_k) \quad \text{et} \quad \sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_k) \tag{1}$$

Autrement dit, x_j est plus proche (ou aussi proche) de x_i que x_k dans l'espace des entrées, mais y_j est moins proche de y_i que y_k dans l'espace des sorties. On parle alors d'une inversion de similarité.

On définit la fonction d'incompatibilité Γ comme le nombre total de triplets incompatibles dans la base :

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) = \left|\left\{(z_i, z_j, z_k) \in \mathcal{CB}^3 \;\middle|\; \sigma_X(x_i, x_j) \geq \sigma_X(x_i, x_k) \wedge \sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_k)\right\}\right|$$

Cette fonction mesure la cohérence globale de la base par rapport au principe de transfert analogique.

Cette formulation permet une implémentation directe et efficace du comptage des incohérences dans la base.

2.3 Prédiction par minimisation de l'incompatibilité

Étant donnée une nouvelle entrée $x_{\text{new}} \in \mathcal{X}$, CoAT prédit la sortie $\hat{y}_{\text{new}}$ en minimisant l'incompatibilité résultante de l'ajout du cas (x_{new}, y) à la base :

$$\text{CoAT}(x_{\text{new}}) = \hat{y}_{\text{new}} = \arg\min_{y \in \mathcal{Y}} \Gamma\left(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{(x_{\text{new}}, y)\}\right)$$

Cette stratégie permet de choisir la sortie la plus cohérente avec la structure des cas déjà observés.

2.4 Algorithmes version naïve

L'algorithme naïf du classifieur CoAT en $\mathcal{O}(n^3)$ proposé dans le document **[Badra, 2020]** consiste à parcourir tous les triplets possibles (z_i, z_j, z_k) dans la base de cas union la nouvelle situation $z_{new} = (x_{new}, y_{new})$. Pour chaque triplet, on vérifie si la contrainte 1 est violée.

Algorithm 1: Algorithme naïf de détection des inversions de similarité

Data: Un ensemble $E = \{(x_i, y_i)\}_{i=1}^{n}$
Result: Nombre total d'inversions de similarité

```
inversions ← 0
for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            if σ_X(x_i, x_j) ≥ σ_X(x_i, x_k) et σ_Y(y_i, y_j) < σ_Y(y_i, y_k) then
                inversions ← inversions + 1
return inversions
```

3 Amélioration algorithmique

Dans cette section nous allons détailler nos travaux qui nous ont permis d'améliorer la complexité du classifieur CoAT.

D'après nos analyses que l'on explicitera juste après, on remarque que les triplets qui ne contiennent la nouvelle situation $z_{new} = (x_{new}, y_{new})$ n'ont aucune utilité pour la prédiction. Ainsi, pour améliorer l'efficacité, cette version optimisée ne teste que les triplets où la nouvelle situation z_{new} intervient, ce qui permet de réduire la complexité à $\mathcal{O}(n^2)$.

Plus précisément, on insère z_{new} dans chacun des rôles possibles dans un triplet (i, j, k), et on boucle sur toutes les paires (i, j) restantes de la base. Cela permet de détecter toutes les inversions de similarité causées par z_{new}, sans recalculer les relations internes à la base déjà existante.

3.1 Analyses mathématiques

La première observation est que la fonction Γ peut être réécrite par une somme d'indicatrice, i.e.

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) = \sum_{(z_i, z_j, z_k) \in \mathcal{CB}^3} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)$$

où

$$\mathbb{I}_{\text{inc}}(z_i, z_j, z_k) = \begin{cases} 1 & \text{si l'équation 1 est satisfaite} \\ 0 & \text{sinon} \end{cases}$$

Cet réécriture est nécessaire / nous permet de simplifier qqch. / de continuer blabla.

Lemme 1. *Soit z_{new} une nouvelle situation. L'ensemble des triplets sur la base augmentée $(CB \cup \{z_{new}\})^3$ peut être décomposé comme suit :*

$$(\mathcal{CB} \cup \{z_{new}\})^3 = \mathcal{CB}^3 \sqcup U$$

où U est l'ensemble des triplets dans lesquels z_{new} intervient au moins une fois.

Démonstration. Comme $\mathcal{CB}$ et $\{z_{new}\}$ sont disjoints, l'ensemble des triplets se décompose en :

$$\begin{aligned}(\mathcal{CB} \cup \{z_{new}\})^3 = \mathcal{CB}^3 \\ \sqcup\ \{z_{new}\}^3 \sqcup\ (\mathcal{CB}^2 \times \{z_{new}\}) \sqcup\ (\mathcal{CB} \times \{z_{new}\} \times \mathcal{CB}) \sqcup\ (\{z_{new}\} \times \mathcal{CB}^2) \\ \sqcup\ (\mathcal{CB} \times \{z_{new}\}^2) \sqcup\ (\{z_{new}\} \times \mathcal{CB} \times \{z_{new}\}) \sqcup\ (\{z_{new}\}^2 \times \mathcal{CB})\end{aligned}$$

On pose alors :

$$\begin{aligned}U = \{z_{new}\}^3 \sqcup\ (\mathcal{CB}^2 \times \{z_{new}\}) \sqcup\ (\mathcal{CB} \times \{z_{new}\} \times \mathcal{CB}) \sqcup\ (\{z_{new}\} \times \mathcal{CB}^2) \\ \sqcup\ (\mathcal{CB} \times \{z_{new}\}^2) \sqcup\ (\{z_{new}\} \times \mathcal{CB} \times \{z_{new}\}) \sqcup\ (\{z_{new}\}^2 \times \mathcal{CB})\end{aligned}$$

□

Proposition 1. *L'ensemble U des triplets contenant la nouvelle situation z_{new} peut être décomposé comme :*

$$U = U_1 \sqcup U_2 \sqcup U_3 \sqcup U_4 \sqcup U_5 \sqcup U_6 \sqcup U_7$$

avec :

$$\begin{aligned}U_1 &= \{(z_{new}, z_j, z_k) \mid (z_j, z_k) \in \mathcal{CB}^2\} \\ U_2 &= \{(z_i, z_{new}, z_k) \mid (z_i, z_k) \in \mathcal{CB}^2\} \\ U_3 &= \{(z_i, z_j, z_{new}) \mid (z_i, z_j) \in \mathcal{CB}^2\} \\ U_4 &= \{(z_{new}, z_{new}, z_k) \mid z_k \in \mathcal{CB}\} \\ U_5 &= \{(z_{new}, z_j, z_{new}) \mid z_j \in \mathcal{CB}\} \\ U_6 &= \{(z_i, z_{new}, z_{new}) \mid z_i \in \mathcal{CB}\} \\ U_7 &= \{(z_{new}, z_{new}, z_{new})\}\end{aligned}$$

Cette proposition vient directement de la décomposition de $(\mathcal{CB} \cup \{z_{new}\})^3$ obtenue précédemment.

Proposition 2. *Soit $z_{new} = (x_{new}, y)$ une nouvelle situation. La fonction d'incompatibilité totale vérifie :*

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\}) = \underbrace{\Gamma(\sigma_X, \sigma_Y, \mathcal{CB})}_{\text{indépendant de } z_{new}} + \sum_{(z_i, z_j, z_k) \in U} \mathbb{I}_{inc}(z_i, z_j, z_k)$$

où $\mathbb{I}_{inc}$ est la fonction indicatrice d'inversion de similarité.

Démonstration. On a par définition :

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\}) = \sum_{(z_i, z_j, z_k) \in (CB \cup \{z_{new}\})^3} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)$$

D'après la décomposition du Lemme 1 :

$$(\mathcal{CB} \cup \{z_{new}\})^3 = \mathcal{CB}^3 \sqcup\ U$$

Ainsi, on peut diviser $\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\})$ en une somme :

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\}) = \Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) + \Gamma(\sigma_X, \sigma_Y, U)$$

On a alors :

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\}) = \Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) + \sum_{(z_i, z_j, z_k) \in U} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)$$

□

Théorème 1. *Soit x_{new} une nouvelle entrée, et soit $\mathcal{Y}$ l'ensemble des valeurs possibles pour sa sortie. Alors l'algorithme CoAT prédit :*

$$\mathrm{CoAT}(x_{new}) = \arg\min_{y\in\mathcal{Y}} \Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\})$$

et cette expression équivaut à :

$$\mathrm{CoAT}(x_{new}) = \arg\min_{y\in\mathcal{Y}} \sum_{(z_i,z_j,z_k)\in U(y)} \mathbb{I}_{inc}(z_i, z_j, z_k)$$

où $U(y)$ est l'ensemble des triplets contenant la situation $z_{new} = (x_{new}, y)$.

Démonstration. Soit $z_{new} = (x_{\text{new}}, y)$ une situation candidate à ajouter à la base $\mathcal{CB}$. D'après la Proposition 2, on peut écrire :

$$\Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{z_{new}\}) = \Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) + \sum_{(z_i,z_j,z_k)\in U(y)} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)$$

où $U(y)$ est l'ensemble des triplets dans lesquels z_{new} apparaît.

Or, la quantité $\Gamma(\sigma_X, \sigma_Y, \mathcal{CB})$ dépend uniquement de la base existante $\mathcal{CB}$ et ne dépend pas du choix de y. On peut donc écrire, pour toute fonction f telle que $f(y) = (C + g(y))$ avec C constante :

$$\arg\min f(y) = \arg\min(C + g(y)) = \arg\min g(y)$$

Finalement comme $\Gamma(\sigma_X, \sigma_Y, \mathcal{CB})$ constante :

$$\begin{aligned}\arg\min_{y\in\mathcal{Y}} \Gamma(\sigma_X, \sigma_Y, \mathcal{CB} \cup \{(x_{\text{new}}, y)\}) &= \arg\min_{y\in\mathcal{Y}} (\ \Gamma(\sigma_X, \sigma_Y, \mathcal{CB}) + \sum_{(z_i,z_j,z_k)\in U(y)} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)\ ) \\ &= \arg\min_{y\in\mathcal{Y}} \sum_{(z_i,z_j,z_k)\in U(y)} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)\end{aligned}$$

Ainsi, le choix de y minimisant $\Gamma(\mathcal{CB} \cup \{(x_{\text{new}}, y)\})$ revient à minimiser uniquement la contribution des triplets contenant z_{new}.

On conclut donc :

$$\mathrm{CoAT}(x_{\text{new}}) = \arg\min_{y\in\mathcal{Y}} \sum_{(z_i,z_j,z_k)\in U(y)} \mathbb{I}_{\text{inc}}(z_i, z_j, z_k)$$

□

Lemme 2. *i) Pour σ_X et σ_Y quelconque*

$$\Gamma(\sigma_X, \sigma_Y, U_6) = \Gamma(\sigma_X, \sigma_Y, U_7) = 0$$

ii) si $\sigma_Y(y_1, y_2)$ atteint son minimum lorsque $y_1 = y_2$ alors :

$$\Gamma(\sigma_X, \sigma_Y, U_5) = 0$$

iii) si $\sigma_Y(y_1, y_2)$ atteint son maximum lorsque $y_1 = y_2$ alors :

$$\Gamma(\sigma_X, \sigma_Y, U_4) = 0$$

avec U_4, U_5, U_6, U_7 défini dans la Définition 1

Démonstration. Rappelons qu'une inversion de similarité a lieu si, pour un triplet (z_i, z_j, z_k), on a :

$$\sigma_X(x_i, x_j) \geq \sigma_X(x_i, x_k) \quad \text{et} \quad \sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_k)$$

Preuve de i) :

— Dans U_6, on a $(z_i, z_{\text{new}}, z_{\text{new}})$, donc $x_j = x_k$ et $y_j = y_k$. Alors :

$$\sigma_X(x_i, x_j) \geq \sigma_X(x_i, x_k) \quad \text{mais} \quad \sigma_Y(y_i, y_j) = \sigma_Y(y_i, y_k) \Rightarrow \text{pas d'inversion.}$$

— Dans U_7, les trois éléments du triplet sont égaux. On a donc :

$$\sigma_X(x_i, x_j) = \sigma_X(x_i, x_k), \quad \sigma_Y(y_i, y_j) = \sigma_Y(y_i, y_k) \Rightarrow \text{pas d'inversion.}$$

Donc $\Gamma(\sigma_X, \sigma_Y, U_6) = \Gamma(\sigma_X, \sigma_Y, U_7) = 0$.

Preuve de ii) : Dans U_5, les triplets sont de la forme $(z_{\text{new}}, z_j, z_{\text{new}})$. On a donc $y_i = y_k$ (les deux correspondent à z_{new}), donc :

$$\sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_k) = \sigma_Y(y_i, y_i)$$

Mais par hypothèse, $\sigma_Y(y_i, y_i)$ est le **minimum**, donc la condition

$$\sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_i)$$

n'est jamais satisfaite. Donc aucune inversion n'est possible, et :

$$\Gamma(\sigma_X, \sigma_Y, U_5) = 0$$

Preuve de iii) : Même raisonnement que pour ii), mais cette fois pour les triplets $(z_{\text{new}}, z_{\text{new}}, z_k) \in U_4$.

Ici, $y_i = y_j$ et donc :

$$\sigma_Y(y_i, y_j) = \sigma_Y(y_i, y_i)$$

Par hypothèse, $\sigma_Y(y_i, y_i)$ est le **maximum**, donc il est impossible que :

$$\sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_k)$$

car le côté gauche est maximal. Donc aucune inversion possible :

$$\Gamma(\sigma_X, \sigma_Y, U_4) = 0$$

□

Proposition 3. *La complexité de l'algorithme optimisé basé sur l'ensemble U est en* $\mathcal{O}(n^2)$.

Démonstration. On rappelle que U est défini comme ceci dans la Définition 1 :

$$U = U_1 \sqcup U_2 \sqcup U_3 \sqcup U_4 \sqcup U_5 \sqcup U_6 \sqcup U_7$$

— D'après le lemme 2 i), $\Gamma(\sigma_X, \sigma_Y, U_6) = \Gamma(\sigma_X, \sigma_Y, U_7) = 0$
Donc ces ensembles n'interviennent pas dans notre algorithme.

— Les sous-ensembles U_4, U_5 contiennent chacun n éléments (on boucle sur un seul élément de $\mathcal{CB}$), donc sont en $\mathcal{O}(n)$.
— Les ensembles U_1, U_2, U_3 contiennent chacun n^2 éléments (on boucle sur deux éléments de $\mathcal{CB}$), donc ils dominent la complexité.

Au final :

$|U| = \mathcal{O}(n^2) \quad \Rightarrow \quad$ La complexité de l'algorithme optimisé basé sur l'ensemble U est en $\mathcal{O}(n^2)$.

□

3.2 Proposition Algorithme optimisé

D'après l'analyse précédente, on propose le pseudo-code ci-dessous pour formaliser la version optimisée du calcul de la complexité pour les ensembles U_1, U_2, U_3 définis précédemment.

Algorithm 2: Calcul de la complexité induite par z_{new} (version optimisée en $\mathcal{O}(n^2)$)

Data: Base de cas $CB = \{(x_i, y_i)\}_{i=1}^{n}$, nouvelle situation $z_{\text{new}} = (x_{\text{new}}, y_{\text{new}})$
Result: Nombre total d'inversions de similarité causées par z_{new}

$cmp \leftarrow 0$ `// Compteur d'inversions`
foreach $(z_i, z_j) \in CB \times CB$, $(i, j) \in [1, n]^2$ **do**
  `// Cas 1 : z_new joue le premier rôle dans le triplet`
  if $\sigma_X(x_{new}, x_i) \geq \sigma_X(x_{new}, x_j)$ ***and*** $\sigma_Y(y_{new}, y_i) < \sigma_Y(y_{new}, y_j)$ **then**
    $cmp \leftarrow cmp + 1$
  `// Cas 2 : z_new joue le second rôle dans le triplet`
  if $\sigma_X(x_i, x_{new}) \geq \sigma_X(x_i, x_j)$ ***and*** $\sigma_Y(y_i, y_{new}) < \sigma_Y(y_i, y_j)$ **then**
    $cmp \leftarrow cmp + 1$
  `// Cas 3 : z_new joue le troisième rôle dans le triplet`
  if $\sigma_X(x_i, x_j) \geq \sigma_X(x_i, x_{new})$ ***and*** $\sigma_Y(y_i, y_j) < \sigma_Y(y_i, y_{new})$ **then**
    $cmp \leftarrow cmp + 1$
return cmp

L'algorithme suivant traduit le fonctionnement de CoAT et renvoie la prédiction associée à une nouvelle entrée.

Algorithm 3: Prédiction de la classe de x_{new} par minimisation de complexité

Data: Base de cas $CB = \{(x_i, y_i)\}_{i=1}^{n}$, nouvelle entrée x_{new}, ensemble des classes possibles $Y = \{y_1, \ldots, y_k\}$
Result: Classe prédite pour x_{new}

$CB_{\text{temp}} \leftarrow CB$
liste_résultats $\leftarrow$ liste vide
foreach $y \in Y$ **do**
  $z_{\text{new}} \leftarrow (x_{\text{new}}, y)$
  $CB_{\text{temp}} \leftarrow CB \cup \{z_{\text{new}}\}$ `// Ajout hypothétique à la base`
  $res \leftarrow$ compute_complexity(CB_{temp})
  Ajouter res à liste_résultats
$indice_{\min} \leftarrow$ indice du plus petit élément de liste_résultats
return $Y[indice_{\min}]$

4 Amélioration par parallélisation

L'algorithme CoAT, même optimisé à $\mathcal{O}(n^2)$, peut devenir trop lent lorsqu'on travaille avec de grandes bases de cas. En Python natif, l'exécution repose uniquement sur le processeur (CPU), et les boucles explicites, même optimisées, entraînent un ralentissement significatif dès que la taille n augmente.

Pour pallier cette contrainte, l'amélioration algorithmique par parallélisation **[Owens, 2008]** s'impose comme une solution efficace. Elle consiste à répartir les calculs sur plusieurs unités de traitement afin d'exploiter la puissance de calcul parallèle des architectures modernes, notamment les GPU (Graphics Processing Units). Ces processeurs, initialement conçus pour le rendu graphique, sont aujourd'hui massivement utilisés en calcul scientifique et machine learning, car ils offrent des milliers de cœurs capables de traiter simultanément de nombreuses opérations.

Pour faciliter l'exploitation de ces architectures puissantes, des bibliothèques spécialisées

comme PyTorch [**Paszke**, **2019**] ou CUDA [**Nickolls**, **2008**] fournissent des outils qui permettent aux développeurs de paralléliser leurs calculs plus simplement. En particulier, PyTorch offre une interface intuitive pour coder des modèles et algorithmes parallélisés, avec une prise en charge native du calcul sur GPU.

Notre objectif est donc de reformuler l'algorithme CoAT en version vectorisée, en remplaçant les boucles par des opérations PyTorch sur GPU, tout en conservant la logique du calcul de complexité.

4.1 Introduction à pytorch

PyTorch repose sur le même principe que NumPy, mais avec la capacité d'exécuter les opérations soit sur CPU, soit sur GPU via CUDA (NVIDIA) à l'aide de la ligne de code suivante :

```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Cette bibliothèque gère des tensors, une structure de données similaire à un tableau (ou matrice), mais généralisée à n dimensions. Tout comme les nparrays, on peut initialiser et faire des opérations directement sur les tensors, ce qui évite de devoir utiliser des boucles tout en pouvant transférer les opérations sur le GPU comme suit :

```
vector = torch.arange(5) // tensors contenant les chiffres de 0    5
vector = vector.to(device) // transfere du tensor sur le GPU
vector_tmp = vector * 2     // -> chaque element de tensor * 2   -> retourne un
    vecteur de meme taille
```

Par ailleurs, on peut faire des opérations entre des tensors qui n'ont pas la même taille ou dimension tant qu'ils sont sur le même appareil. Par exemple :

```
vector_tmp = vector + torch.tensor([1, 2, 3, 4, 5]).to(device)
```

Le paramètre dim permet d'appliquer des opérations (comme sum, mean, max, etc.) le long d'un axe spécifique du tenseur, ce qui est essentiel pour manipuler correctement les structures de données complexes. Le code suivant illustre cette fonctionnalité :

```
x = torch.tensor([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
])

print(torch.sum(x, dim=0))  #     somme colonne par colonne
# tensor([5., 7., 9.]) => [1+4, 2+5, 3+6]

print(torch.sum(x, dim=1))  #     somme ligne par ligne
# tensor([6., 15.]) => [1+2+3, 4+5+6]
```

Grâce à ces mécanismes, PyTorch permet d'écrire du code vectorisé, lisible et optimisé, sans boucle explicite, ce qui est indispensable pour le traitement de données en masse.

4.2 Passage du calcul en boucle au calcul parallèle

L'implémentation de CoAT avec Pytorch permet de réduire les boucles des implémentation classique, en changeant le types des données des paramètres des fonctions, en particulier pour la fonction qui compte le nombre d'incompatibilité. La table 1, nous montre une traduction des codes utiles à l'implémentation du classifieur.

Soit $X \in \mathbb{R}^{n \times d}$ un tableau représentant les n exemples de la base, chaque ligne x_i étant une entrée de dimension d.

On note :

— `sim_X[i,j]` $= \sigma_X(x_i, x_j)$: une mesure de similarité entre entrées ;

— `sim_Y[i,j]` = $\sigma_Y(y_i, y_j)$: une mesure de similarité entre sorties (ou étiquettes) ;
— `triplets` : un tableau d'indices de forme $(m, 3)$ contenant m triplets (i, j, k) à évaluer.

Boucle en Python natif	**Version vectorisée avec PyTorch**

`for i in range(n):`
`  for j in range(n):`
`  dist[i][j] = X[i] - X[j]`
	`X_torch = torch.tensor(X)`
	`dist = torch.cdist(X_torch, X_torch)`

`gamma = 0`
`for i, j, k in triplets:`
`  if sim_X[i,j] >= sim_X[i,k] and \`
`  sim_Y[i,j] < sim_Y[i,k]:`
`  gamma += 1`
	`cond_X = sim_X[:,:,0] >= sim_X[:,:,1]`
	`cond_Y = sim_Y[:,:,0] < sim_Y[:,:,1]`
	`gamma = (cond_X & cond_Y).sum()`

TABLE 1 – Comparaison entre des implémentations classiques en Python et leurs équivalents vectorisés en PyTorch.

Analysez les complexités temporelle et spatiale.

5 Expérimentations

L'objectif de cette section est de vérifier de manières concrètes à l'aide de tests, nos analyses effectuées dans les parties précédentes. On dispose pour cela d'un GPU *NVIDIA GeForce RTX 3050* avec 2560 CUDA cœurs (d'après [2]) qui nous permettra de faire nos tests pour la version PyTorch. Les versions Python utiliseront quant à elles, seulement le CPU.

Application choisie : classification binaire de points dans le plan

Dans le cadre de ce projet, nous avons choisi une application simple de la méthode CoAT, permettant une visualisation intuitive et un contrôle direct des données.

Chaque situation $z_i = (x_i, y_i)$ est un point dans le plan $\mathbb{R}^2$:
— l'entrée $x_i \in \mathbb{R}^2$ est représentée par les coordonnées du point,
— la sortie $y_i \in \{\text{rouge}, \text{bleu}\}$ est une étiquette de **classe** (couleur).

Les fonctions de similarité utilisées sont définies comme suit :
— **Similarité sur les entrées** :

$$\sigma_X(x_i, x_j) = \frac{1}{1 + \|x_i - x_j\|_2}$$

où $\|x_i - x_j\|_2$ désigne la distance euclidienne entre x_i et x_j dans $\mathbb{R}^2$. Cette fonction est maximale pour des points proches et tend vers 0 à mesure que la distance augmente.
— **Similarité sur les sorties** :

$$\sigma_Y(y_i, y_j) = \begin{cases} 1 & \text{si } y_i = y_j \\ 0 & \text{sinon} \end{cases}$$

Il s'agit d'une fonction booléenne qui indique si deux points ont la même couleur.

Dans ce cadre, le classifieur CoAT, confronté à un nouveau point $x_{\text{new}} \in \mathbb{R}^2$, prédit la couleur $y \in \{\text{rouge}, \text{bleu}\}$ qui minimise le nombre d'**inversions de similarité** (i.e. de triplets incompatibles) dans la base étendue par (x_{new}, y).

Cette application permet d'évaluer les performances de CoAT sur un problème de **classification binaire géométrique**, tout en facilitant l'analyse visuelle des prédictions.

Protocole expérimental

Nous décrivons ici le protocole choisi pour effectuer les tests lors de ce projet.

Notre protocole est le suivant :

- On génère la base de cas de la manière suivante : les points rouges sont générés de manière aléatoire dans un cercle séparément des points bleus.
- On génère un ensemble contenant les coordonnées aléatoire sur le plan de point à tester (sans couleur).
- On effectue une série de test pour chaque implémentation en faisant varier le nombre de points dans la base de cas (CB) avec des mesures de similarités fixes (décrites dans la section précédente).
- On effectue ces tests sur plusieurs points et on fait une moyenne des temps obtenus, que nous affichons dans un graphique afin d'avoir une représentation visuelle de nos résultats.

5.1 Implémentation

Cette partie décrit la structure de notre code.

5.1.1 Fonctions communes aux versions en Python natif

Certaines fonctions sont communes aux deux versions, car elles correspondent aux définitions fondamentales du modèle :

- `similarityX(x1, x2)` : mesure la similarité entre deux entrées x_1 et x_2, comme défini dans la section précédente.
- `similarityY(y1, y2)` : mesure la similarité entre deux sorties y_1 et y_2.
- `coat_predict(x_new, CB, ys_unique)` / `coat_predict_optimized(x_new, CB, ys_unique)` : dans les deux cas, la fonction prédit la meilleure valeur de y pour un nouveau point x_{new} en testant chaque classe possible et en sélectionnant celle qui minimise la complexité.

Les données sont représentées sous forme de liste de couples (x_i, y_i), chaque couple formant une observation de la base de données.

5.1.2 Version naïve en Python

L'implémentation de la version naïve de CoAT en Python repose sur 5 fonctions principales :

- Les fonctions décrites dans la section précédente.
- `indexes_triplet(data)` : renvoie tous les triplets d'indices possibles (i,j,k) dans data. Cette fonction utilise yield au lieu de return afin de créer un générateur qui produit à la volée toutes les combinaisons possibles des triplets (i, j, k). Ce choix permet d'éviter de stocker l'ensemble des coordonnées en mémoire, ce qui rend la fonction plus légère et efficace, notamment lorsque le nombre de combinaisons devient important.
- `compute_complexity(data)` : calcule la complexité en comptant les inversions de similarité en parcourant tous les triplets possibles.

Les données sont représentées sous la forme de listes de tuples (x_i, y_i) pour la version Python classique.

5.1.3 Version optimisée en Python

L'implémentation de la version optimisée de CoAT en Python repose sur 6 fonctions :

- Les fonctions communes avec la version précédente.
- `indexes_couple(data)` : renvoie tous les couples d'indices possibles (i,j) dans data. Cette fonction utilise à nouveau yield.
- `evaluate_triplet_inversion(data1,data2,data3)` : Fonction intermédiaire permettant de faciliter le calcul des inversions de similarité.
- `compute_complexity_optimized(data,z_new))` : calcule la complexité en comptant les inversions de similarité en ne parcourant que les cas où z_{new} intervient.

5.1.4 Version PyTorch

Pour l'implémentation de la version en PyTorch, nous avons utilisé une fonction optimisée de distance euclidienne entre tous les couples (x_i, x_j), en utilisant la formule :

$$\|x_i - x_j\|^2 = \|x_i\|^2 + \|x_j\|^2 - 2\langle x_i, x_j \rangle$$

Ainsi, notre version PyTorch repose sur 5 fonctions principales :

- `optimized_euclidean_distance(A, B)` :renvoie la distance euclidienne optimisée pour PyTorch.
- `sim_T_CB_1(T_X: torch.Tensor, CB_X: torch.Tensor) -> torch.Tensor` : calcule la distance euclidienne au carré entre chaque point de T et chaque point de CB en utilisant la fonction optimisée.
- `sim_T_CB_2(T_Y: torch.Tensor, CB_Y: torch.Tensor) -> torch.Tensor` : calcule la similarité de couleur entre les points de T et CB.
- `compute_incompatibility(sim_T_CB_1: torch.Tensor, sim_T_CB_2: torch.Tensor) -> torch.Tensor)` : calcule l'incompatibilité pour chaque point du tensor T.
- `predict_CoAT(T: torch.Tensor, CB: torch.Tensor, Q: list) -> torch.Tensor)` : prédit la couleur de chaque point de T selon la méthode du classifieur COAT.

5.2 Résultats et expérimentations

5.2.1 Illustration de la prédiction

Les deux figures suivantes montrent l'effet de la prédiction par l'algorithme CoAT sur un ensemble de points.

À gauche, les points noirs représentent les éléments dont l'étiquette est inconnue. À droite, ces mêmes points ont été prédits et colorés selon la classe attribuée. On observe ainsi l'impact direct de la prédiction sur la configuration du plan.

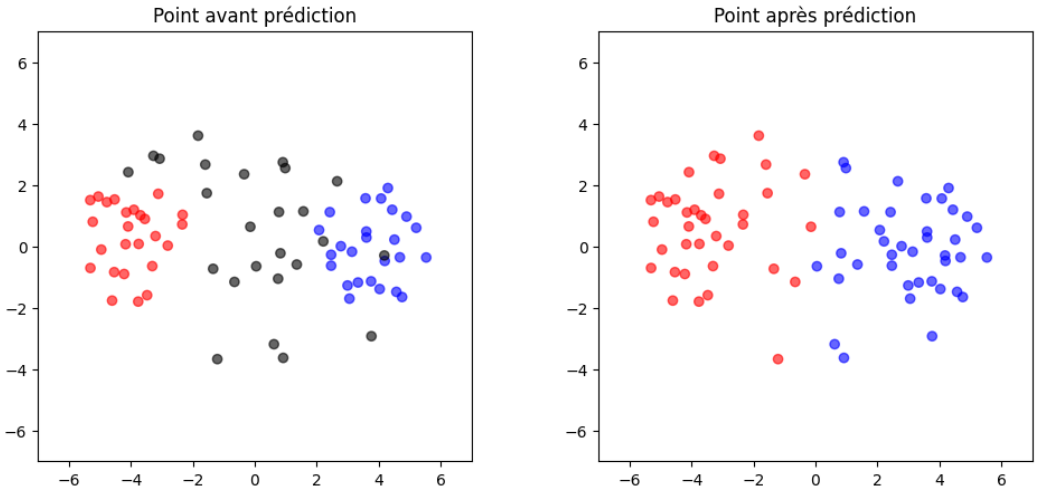

FIGURE 1 – Prédiction réalisée par CoAT sur un ensemble de point

5.2.2 Comparaison versions Pythons

Dans cette partie, nous analysons les temps de calcul pour nos 2 versions Python, sur des bases de cas de tailles différentes. La figure 2 illustre cela ci-dessous.

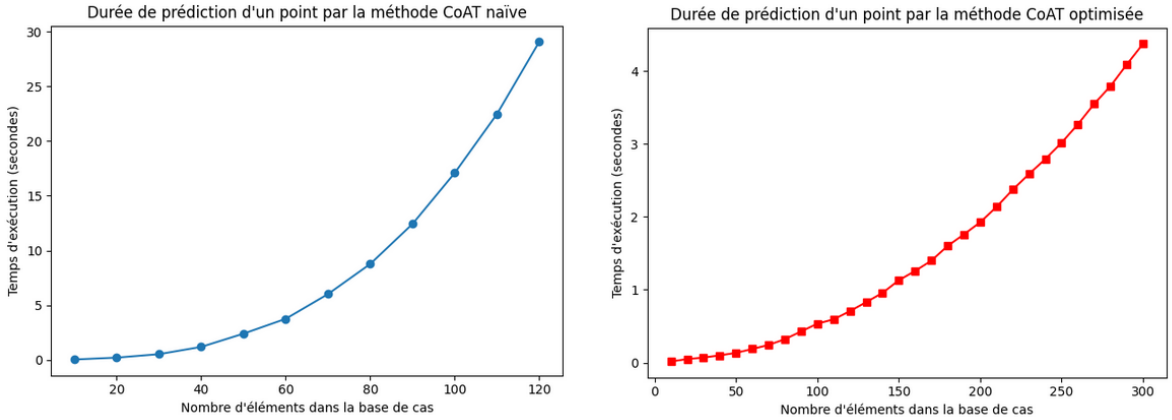

FIGURE 2 – Temps d'exécution des deux versions en fonction de la taille de CB

Version naïve Pour cette version, le temps de calcul devient rapidement très important, environ 15 secondes pour prédire un point dans une base de cas de taille 100. Ceci s'explique par la complexité de la version naïve en $\mathcal{O}(n^3)$. Cette méthode devient peu efficace sur des grandes bases de cas et lorsqu'on souhaite prédire un nombre de points conséquents avec des temps de calcul pouvant atteindre des dizaines de minutes, voire des heures.

Version optimisée Pour cette version, le temps de calcul pour prédire un point sur une base de cas de taille de 100 points est d'environ 0.5 secondes, ce qui représente une amélioration d'un facteur 30. Cette version représente donc déjà un réel avantage par rapport à la version naïve. Elle permet d'étudier des cas plus importants avec des temps d'exécution qui restent raisonnables. Néanmoins, sur des bases de cas plus conséquentes, comme pour une base de 300 points, le temps de prédiction grimpe à 4,5 secondes. Ce résultat reste acceptable pour des traitements simples, mais devient problématique dans des contextes de grande échelle où les temps de calcul peuvent atteindre des dizaines de minutes.

5.2.3 Version PyTorch

Tests sur différentes bases de cas

Dans un premier temps, nous nous demandons, comme dans les cas précédents, comment évolue le temps de calcul lorsque la taille de la base de cas varie. Sur PyTorch, l'utilisation du GPU, permettant plus de calculs que le CPU, nous avons pu effectuer des tests à des échelles supérieures.

Le graphique ci-dessous illustre l'évolution du temps de calcul nécessaire à la prédiction de 100 points, en fonction de la taille de la base de cas, jusqu'à un maximum de 300 éléments.

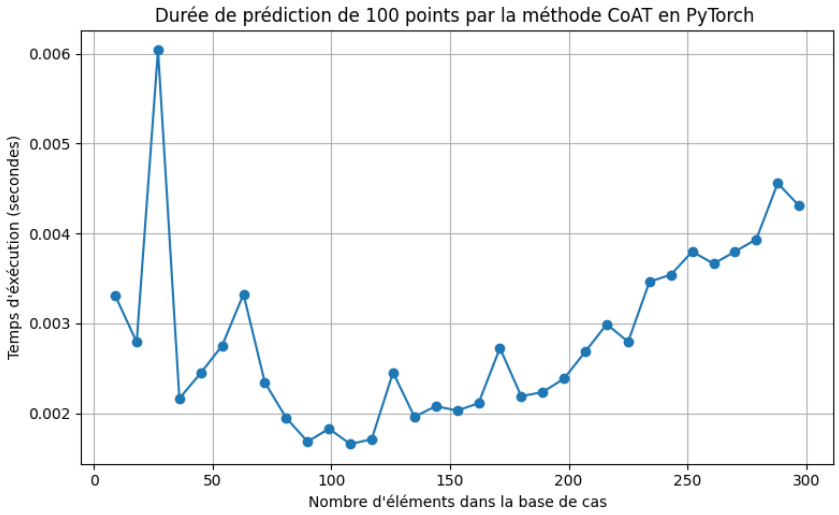

FIGURE 3 – Temps d'exécution de la version avec Pytorch en fonction de la taille de CB

Comparativement à une implémentation naïve, qui nécessite environ 15 secondes pour prédire un unique point à partir d'une base de cas de 100 éléments d'après la figure 2, la version utilisant PyTorch permet de prédire 100 points en environ 0.004 secondes pour une base de même taille. Cette amélioration correspond à un gain de performance d'un facteur 3750, tout en augmentant le nombre de points prédits par un facteur 100.

Pour la version optimisée de CoAT en Python, le temps de calcul est d'environ 0.5 seconde pour prédire 1 point sur une base de cas de 100 éléments d'après la figure 2. Bien qu'elle reste plus rapide que l'implémentation naïve, cette version reste toutefois environ 125 fois plus lente que l'implémentation vectorisée sur PyTorch.

Ces résultats mettent en évidence l'efficacité du traitement vectorisé et de l'exécution parallèle sur GPU dans l'approche optimisée.

Néanmoins, il nous est ici impossible de déduire une quelconque augmentation du temps de calcul pour la version Pytorch

Tests lorsque la base de cas et le nombre de points à prédire varient

Dans cette partie, nous nous demandons ce qu'il se passe lorsque la taille de la base de cas augmente, mais aussi lorsque le nombre de points à prédire augmente. En effet, la capacité de calcul offerte par la version vectorisée en PyTorch nous permet d'envisager des tests sur des bases nettement plus conséquentes.

La figure 4 illustre l'évolution du temps de calcul nécessaire lorsque la taille de la base de cas et le nombre de points à prédire augmentent.

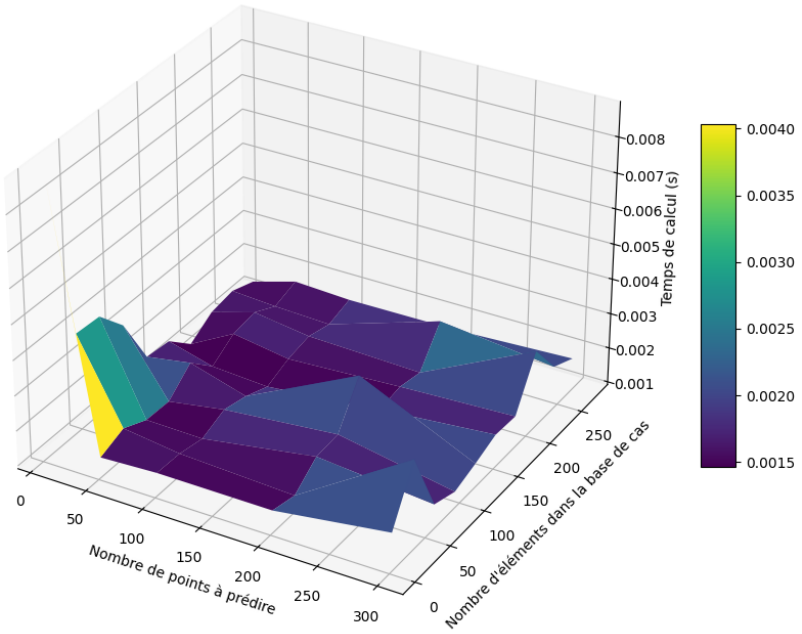

FIGURE 4 – Temps d'exécution de la version Pytorch en fonction de la taille de CB et du nombre de point à prédire

Malgré l'augmentation conjointe de la taille des bases de cas et du nombre de points à prédire, le temps de calcul reste quasiment constant dans la version PyTorch, contrairement à l'implémentation naïve en Python où une croissance significative est observée. Ce comportement est non seulement attendu, mais aussi caractéristique des architectures parallèles, notamment des GPU. Plusieurs phénomènes expliquent ce résultat :

- **Parallélisme massif du GPU** : Les GPU sont conçus pour exécuter un grand nombre d'opérations simultanément. Tant que la taille des données reste dans les limites de la mémoire disponible et que des unités de calcul restent inactives, l'ajout de points supplémentaires ne provoque qu'une charge marginale supplémentaire, répartie sur les cœurs disponibles.
- **Effet de saturation (ou effet de plateau)** : Pour des tailles de données modérées, le GPU est souvent sous-exploité. Dans cette phase, l'augmentation du nombre de points ou de la base de cas n'entraîne que peu, voire pas du tout, de variation du temps de calcul. Ce n'est qu'au-delà d'un certain seuil — lorsque la mémoire est saturée ou que les unités de calcul sont toutes sollicitées — que le temps d'exécution croît de façon marquée.
- **Optimisation vectorielle et calcul matriciel** : Les opérations fondamentales utilisées dans l'algorithme (produits scalaires, normalisations, comparaisons matricielles, etc.) sont implémentées de manière vectorisée et optimisée sur GPU. Le coût de ces opérations ne croît pas linéairement avec le nombre d'éléments comme dans les implémentations séquentielles.

Ainsi, la stabilité observée du temps de calcul n'est pas une anomalie, mais une conséquence directe de la nature massivement parallèle de l'environnement d'exécution GPU.

6 Limites et évolutions possibles

6.1 Limites du calcul parallèle avec PyTorch

Bien que l'approche vectorisée et parallélisée via PyTorch permette d'obtenir des performances remarquables, elle n'est pas exempte de limitations. L'une des principales contraintes observées dans ce cadre est liée à la **saturation de la mémoire GPU**. En effet, lorsque la taille de la base de cas devient trop importante, les matrices manipulées deviennent trop volumineuses pour le GPU, entraînant un ralentissement brutal des calculs voire un échec de l'exécution ("out of memory").

Ce phénomène se manifeste généralement après un **effet de palier** : tant que les données tiennent intégralement dans la mémoire GPU, le temps de calcul reste presque constant. Cependant, au-delà d'un certain seuil critique, le coût explose brutalement. Cela correspond au point où la parallélisation ne suffit plus à compenser la surcharge induite par la taille des matrices.

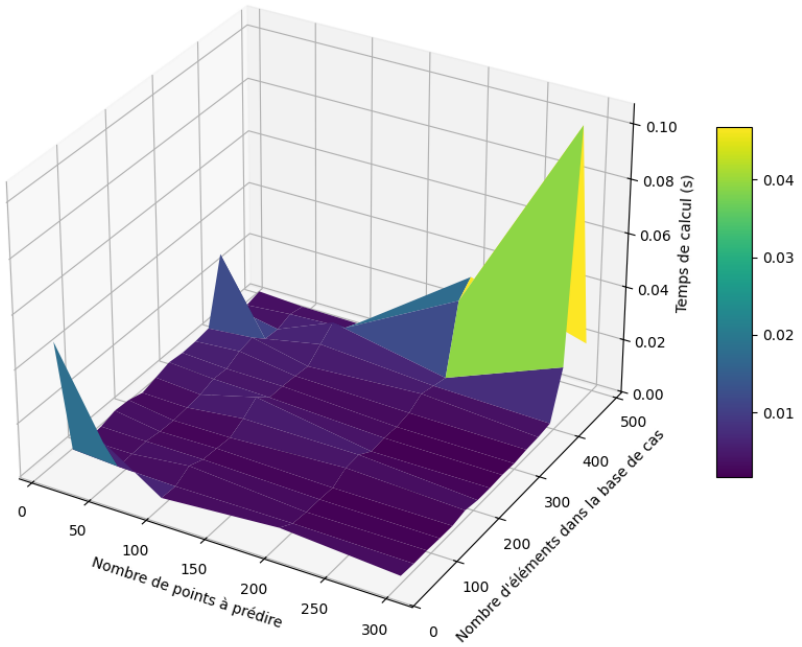

FIGURE 5 – Temps d'exécution de la version avec Pytorch en fonction de la taille de CB et du nombre de point à prédire avec saturation du GPU

Le graphique 5 permet de visualiser le point de rupture où l'approche cesse d'être efficace, mettant ainsi en évidence les limites intrinsèques du calcul parallèle même dans des environnements GPU performants. Ici, le temps de calcul grimpe à 0.1 secondes pour une base de cas de 500 points lors de la prédiction de 300 points ; soit une multiplication par 25 du temps de calcul par rapport au temps d'exécution lorsque le GPU n'est pas surchargé.

6.2 Améliorations possibles

Pour contourner la limitation liée à la mémoire GPU dans l'approche PyTorch, une stratégie d'amélioration que nous avons envisagée consiste à diviser dynamiquement les tenseurs en sous-blocs plus petits. L'idée est de traiter les données par segments compatibles avec les capacités mémoire du GPU.

Plus précisément, on choisit un entier n bien adapté à la configuration matérielle (c'est-à-dire à la mémoire disponible sur le GPU), puis on découpe les matrices ou tenseurs concernés en

n parties égales ou presque. Chaque bloc est alors traité séquentiellement sur le GPU. Cette méthode permet de rester en deçà du seuil critique de saturation mémoire, tout en conservant les avantages du calcul parallèle pour chaque sous-bloc.

Cette approche introduit un facteur de multiplication du temps de calcul d'environ n, puisque les blocs sont traités les uns après les autres. Toutefois, cette augmentation reste largement acceptable dans la pratique, car les opérations vectorisées de PyTorch sur GPU sont très rapides et efficaces même lorsqu'elles sont répétées plusieurs fois.

Ce compromis permet de maintenir une exécution stable et rapide sur des bases de cas de grande taille, tout en évitant les erreurs de type `out of memory`.

7 Conclusion

Synthèse du projet

Ce projet nous a permis d'explorer en profondeur la méthode de classification CoAT (Complexity-based Analogical Transfer), fondée sur la minimisation d'incompatibilités entre similarités d'entrée et de sortie. Après avoir implémenté une version naïve de complexité cubique ($\mathcal{O}(n^3)$) présentée dans **[Badra, 2020]** , nous avons proposé une version optimisée réduisant cette complexité à $\mathcal{O}(n^2)$ grâce à une analyse des triplets pertinents. Enfin, nous avons mis en œuvre une version parallélisée à l'aide de la bibliothèque PyTorch, tirant parti des capacités du calcul vectoriel sur GPU pour atteindre des performances nettement supérieures.

Ce qui a été appris

Ce projet nous a permis de mobiliser et d'approfondir des compétences à la fois théoriques et pratiques :

- En algorithmique, en identifiant les redondances dans les triplets et en prouvant mathématiquement l'amélioration de complexité.
- En implémentation, en traduisant les principes mathématiques en code Python, puis en version vectorisée avec PyTorch.
- En analyse expérimentale, en construisant un protocole de tests rigoureux et en mesurant l'impact des optimisations sur les temps de calcul.
- En gestion de ressources, notamment mémoire GPU, en observant les effets de saturation et en concevant des solutions pour les contourner.

Perspectives futures

Plusieurs pistes pourraient être explorées à l'issue de ce travail :

- Une gestion dynamique des blocs dans la version PyTorch pour s'adapter automatiquement à la mémoire disponible.
- L'introduction d'un apprentissage de métriques de similarité pour adapter σ_X et σ_Y aux données.
- La comparaison avec d'autres classifieurs (comme K-NN [1], SVM) sur des bases réelles ou bruitées.

En conclusion, ce projet a été une excellente opportunité de croiser modélisation mathématique, programmation performante et analyse expérimentale, dans une problématique réelle issue du machine learning.

Références

[1] Fadi Badra, *A Dataset Complexity Measure for Analogical Transfer*, International Joint Conference on Artificial Intelligence (IJCAI), 2020.

[2] Janet L. Kolodner, *An introduction to case-based reasoning*, Artificial Intelligence Review vol 6, 1992.

[3] Todd R. Davis and Stuart J. Russell, *A Logical Approach to Reasoning by Analogy*, IJCAI Int. Jt. Conf. Artif. Intell, 1987.

[4] Adam Paszke *et al.*, *PyTorch : An Imperative Style, High-Performance Deep Learning Library*, Advances in Neural Information Processing Systems (NeurIPS), 32, 2019.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, *GPU computing*, Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, 2008.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with CUDA*, Queue, vol. 6, no. 2, pp. 40-53, 2008.

[7] PyTorch Developers, *CUDA semantics — PyTorch documentation*, `https://pytorch.org/docs/stable/notes/cuda.html`, 2023.

[8] PyTorch Team, *PyTorch Profiler Tutorial*, `https://pytorch.org/tutorials/intermediate/profiler_tutorial.html`, 2023.

[9] Meta AI Research, *Performance Tuning Guide for PyTorch*, `https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html`, 2023.

[10] NVIDIA Corporation, *CUDA C Programming Guide*, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2023.

[11] NVIDIA Corporation, *cuBLAS Library User Guide*, `https://docs.nvidia.com/cuda/cublas/index.html`, 2023.

[12] NVIDIA Developer Tools, *CUDA Occupancy Calculator Documentation*, `https://docs.nvidia.com/cuda/occupancy-calculator/`, 2022.

[13] NVIDIA, *NVIDIA GeForce RTX 3050 Specifications*, `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3050/`, 2022.

Code source

Le code source du projet est disponible sur GitHub à l'adresse suivante :
`https://github.com/alexlcode69/Projet-CoAT`