Projet *Machine Learning*
Régression symbolique – Groupe 3

Une coloration indique l'état d'avancement des différentes propositions avancées :
fait ;
en cours ;
prévu, confiance dans la méthode à mettre en œuvre ;
aucune idée de la méthode à mettre en œuvre/doute sur la pertinence de la proposition ;
sans objet

1 Contexte

1.1 Position du problème : la régression symbolique

Supposons que l'on dispose – par exemple à la suite de l'étude expérimentale d'un système physique quelconque – d'un jeu de variables $(x_1, x_2, ..., x_n, y_1, ..., y_p)_j$. On souhaite comprendre le fonctionnement de ce système sous la forme d'une ou plusieurs relations entre ces différentes variables. On peut, par exemple, vouloir exprimer les p dernières variables $(y_1, ..., y_p)$, regroupées dans la *sortie* $\mathbf{y} \equiv \begin{pmatrix} y_1 \\ ... \\ y_p \end{pmatrix}$, selon les n premières, qui forme les éléments de l'*entrée* $\mathbf{x} \equiv \begin{pmatrix} x_1 \\ ... \\ x_n \end{pmatrix}$:

$$\mathbf{y} = f(\mathbf{x})$$

Il se peut que nous ignorions tout de la physique du système, et que nous espérions que le jeu de données nous informe sur la forme de la fonction f qui la contient. On dira que l'on *résout le problème* $(\mathbf{x}, \mathbf{y})$ lorsque l'on détermine la forme de la fonction f telle que, avec une précision ϵ satisfaisante, on peut écrire :

$$\mathbf{y} = f(\mathbf{x}) + \mathbf{e}, \|\mathbf{e}\| \leq \epsilon.$$

Dans la suite on ne s'intéressera qu'à des problèmes où $p = 1$, le cas $p > 1$ pouvant être reformulé comme l'assemblage de p problèmes à $p = 1$ indépendants.

1.2 Implémentation et pertinence du *machine learning*

Moyennant de bonnes représentations, autant pour les éléments $(\mathbf{x}, \mathbf{y})$ que pour la fonction f les reliant, on peut espérer entraîner un modèle à retrouver l'expression de f dans un jeu de données d'entraînement de la forme $\{(\mathbf{x}, \mathbf{y})_j\}$. Ce jeu de données devra être de caractéristiques similaires aux problèmes que l'on cherchera à résoudre.
Évidemment, pour que le résultat final soit exploitable pour un opérateur humain, il nous faudra spécifier la forme que pourra prendre la fonction f. Typiquement, celle-ci pourra composer, sommer, multiplier :

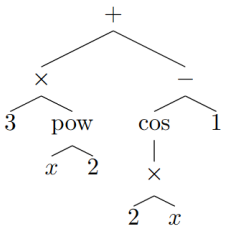
- les variables $(x_1, x_2, ..., x_n)$
- des opérateurs unaires $\Gamma : u \mapsto v$ associant un vecteur à un vecteur.
  Exemple : l'opérateur opposé, noté opp, renvoie l'opposé d'un nombre. L'opérateur cosinus, noté cos, renvoie le cosinus d'un nombre.

— des opérateurs binaires $\lambda : u, v \mapsto w$ associant un vecteur à une paire de vecteurs.
Exemple : l'opérateur somme, noté $+$, renvoie la somme terme à terme de deux vecteurs de la même taille.
— ...

1.2.1 Représentation d'une expression algébrique sous forme d'un arbre

Les expressions mathématiques peuvent être représentées sous forme d'arbres, les opérateurs et les fonctions étant des nœuds internes, les opérandes comme enfants, et les nombres, constantes et variables comme feuilles. Un exemple ci-contre.

$3x^2 + \cos(2x) - 1$



1.2.2 Représentation machine du problème

On pourra utiliser des architectures de type `seq2seq`, ie. prenant en entrée une liste $(x_1, x_2, ..., x_n, y)_j$ de *variables d'entrée* et renvoyant sous la forme d'une liste l'expression de la fonction f. Convenant de l'arité des opérateurs utilisés et de règles de précédence, la notation préfixée (ou postfixée) permet d'exprimer de façon univoque un arbre sous la forme d'une séquence. L'expression prise en exemple est, sous forme de liste : $[+, *, 3, \text{pow}, x, 2, -, \cos, *, 2, x, 1]$.

2 Fabrication d'un jeu de données

Il s'est essentiellement agi d'implémenter la méthode décrite dans [1]. On pourra également travailler en parallèle sur des jeux de données de référence, tels celui fabriqué dans [2], ou ceux du SRbench.

2.1 Caractéristiques du jeu de données

2.1.1 Pertinence

Dans la perspective d'utilisation du modèle pour résoudre des problèmes liés au monde réel, les expressions utilisées pour son entraînement devront être représentatives de celles susceptibles d'être rencontrées dans ses applications. Une grande diversité d'expressions ouvre plus de possibles mais au prix d'expressions bien plus lourdes à manipuler, et d'une durée d'assimilation croissant possiblement exponentiellement pour le modèle.

1. Taille n du vecteur d'entrée [1]
2. Taille des arbres : borne supérieure sur le nombre de nœuds internes.
3. Structure des arbres. Faut-il, par exemple, qu'à nombre de noeuds donnés, l'arbre le plus *left*(resp. *right*)-*leaning*, le plus profond et le plus lage apparaissent avec la même probabilité ?
4. Quel jeu minimal d'opérateurs unaires et binaires employer ?
   Vaut-il mieux introduire un opérateur opp associant à une opérande son opposé, ou bien comprendre l'opposition comme une multiplication par -1 ? Considérer uniquement les lois de puissance entières les plus courantes ($2 > 3$), ou bien introduire un opérateur générique d'exponentiation, au risque de se retrouver avec des tas de solutions en $^{1.97}$?

On pourra répondre à ces questions sur la base de jeux de référence tels que celui de [2], ou ceux du SRbench.

2.1.2 Représentativité

Ayant déterminé ses ingrédients (point précédent : la dimension n du vecteur d'entrée, bornes sur le nombre de nœuds internes, fonctions utilisables, ...), trouver un moyen de déterminer quelle taille de jeu de données d'entraînement est suffisante pour rendre le modèle "performant" sur les problèmes susceptibles d'être rencontrés.

La limitation de nos capacités de calcul implique de traiter itérativement les deux points Pertinence et Représentativité.

1. Pour une preuve de concept on pourra dans un premier temps le fixer à 1.

2.2 Construction de fonctions à l'aide d'arbres de décision

Méthode tirée de [1] :

1. Tirer la dimension n de l'entrée $\mathbf{x}$.
2. Tirer le nombre b d'opérateurs binaires. Tirer b opérateurs binaires.
3. Construire un arbre binaire à b nœuds.
4. Placer sur chaque feuille une variable x_j réelle.
5. Tirer le nombre u d'opérateurs unaires u_r. Tirer u opérateurs binaires. Les placer aléatoirement dans l'arbre.
6. Opérer une transformation affine aléatoire de chaque variable x_j et opérateur unaire u_r.

On convertit ensuite ces arbres sous forme de listes. La fonction `simplify` du module `simpy` simplifie les expressions générées. On parlera de *listes réduites*, d'*arbres réduits*, d'*expressions réduites*. En principe il existe une bijection entre les listes réduites (ou, de façon équivalente, les arbres réduits) et les expressions réduites.

2.3 Échantillonnage de l'espace des variables et fabrication du jeu de données

Méthode d'après [3] :

1. on fixe $k \in [\![0, k_{\max}]\!]$ de barycentres que dont on tire la position $\mathbf{x}_k$ selon $\mathcal{N}(0, 1)$.
2. Pour chacune d'entre elles on tire une variance v_k. On tire ensuite une certaine fraction du nombre total de points N selon $\mathcal{N}(\mathbf{x}_k, \sqrt{v_k})$.
3. On applique à chacune des k sous-distributions une transformation de Haar.
4. On normalise la distribution somme des k sous-distributions transformées, de sorte que sa moyenne soit 0 et sa variance 1.

On effectue un tel tirage d'ensemble de variables pour chaque fonction construite. On ignore toute fonction s'avérant non-évaluable en un des points tirés (on discutera cette stratégie).

Le jeu de données finalement obtenu $\{\{(\mathbf{x}^{(i)}, f(x^{(i)})_j\}_i\}$ pourra utilement être divisé (selon i) en un sous-ensemble d'entraînement et un sous-ensemble d'évaluation.

2.4 Caractérisation du jeu de données

Voir comment le jeu de données finalement produit est typique des Caractéristiques imposées.
Justifier la nécessité d'une méthode plus évoluée qu'une construction récursive des arbres (à chaque nœud choisir avec certaines probabilités, fixées, si l'on place un unaire, un binaire, ou une feuille (variable).
Le rejet des *fonctions* dont l'évaluation en un point renvoie une erreur pourrait tendre à sous-représenter soustractions, de racines... D'autres approches, comme le rejet du point défaillant plutôt que de la fonction, permettent un meilleur échantillonage et renseignent le modèle quant au domaine de définition des fonctions (est-ce une bonne chose ?)

3 Perspectives futures

Maintenant que nous avons un jeu de données regroupant des expressions mathématiques aussi variées que possible, ainsi qu'un échantillonage de leur courbe associée, nous voulons entraîner une IA afin qu'à partir d'un nouveau jeu de points $(\mathbf{x}, \mathbf{y} = f(\mathbf{x}))$ on estime l'expression mathématique de f. Pour ce faire nous avons choisi tout d'abord une méthode moins efficace mais facilement implémentable consistant en un *Fully Connected Neural Networks* (FCNN) classique dont les fonctions d'activation seraient les opérateurs binaires et unaires donnés précédemment, comme écrit dans [4]. La deuxième méthode, plus efficace, repose sur l'article [3], et consiste, à la manière des LLM, à utiliser un *Transformer* pour entraîner notre IA.

Enfin, en utilisant des métriques telles que le score R^2 [2] ou la tolérance de précision [3] nous pouvons évaluer la capacité de chacune des méthodes à prédire une expression juste et cohérente.

3.1 Première approche : réseau dense

La première approche consiste à utiliser un FCNN un peu modifié afin de pouvoir tirer de la régression une expression mathématique utilisable et non une composée de ReLU comme les réseaux classiques nous donneraient. Pour ce faire, on suit la méthode proposé dans [4] : à chaque layer du réseau on applique une transformation affine en multipliant par les poids et en additionnant du bruit, puis on applique une fonction d'activation dont la nature dépend de la position du noeud.
Formellement, au noeud l on obtient pour valeur temporaire :

$$z^{(l)} = W^{(l)} y^{(l-1)} + b^{(l)}$$

Où $W^{(l)}$ est la matrice de poids et $b^{(l)}$ le vecteur de bruit.
On applique ensuite u opérateurs unaires f_i (choisis uniformément parmi $id, \cos, \sin$ et sigm pour éviter des problèmes d'intervalle de définition) et v opérateurs binaires g_i à ces valeurs temporaires pour obtenir la sortie du layer l :

$$\begin{aligned} y^{(l)} =& (f_1(z_1^{(l)}), ..., f_u(z_u^{(l)}), \\ & g_1(z_{u+1}^{(l)}, z_{u+2}^{(l)}), ..., g_v(z_{u+2v-1}^{(l)}, z_{u+2v}^{(l)})) \end{aligned}$$

Pour choisir ensuite nos paramètres, on minimise une *loss function* de type L_2 avec une régularisation L_1 dont le coefficient change en fonction de l'époque (il est nul au début et à la fin de l'apprentissage) :

$$L = \frac{1}{d} \sum_{i=1}^{d} ||y_i - y_i^{(h)}|| + \lambda \sum_{l=1}^{h} |W^{(l)}|_1$$

Pour la minimisation, on peut utiliser une descente de gradient stochastique classique.
Enfin, pour ce problème spécifique nous sommes aussi intéressé par le fait que nous voulons obtenir les expressions mathématiques les plus simples possible (en gardant à l'esprit que celles-ci sont supposées décrire des phénomènes physiques) il nous faut donc un critère supplémentaire pour choisir nos paramètres : celui de la *scarcity* s qui correspond au nombre de noeud actifs dans le réseau :

$$s = \sum_{l=1}^{h} \sum_{i=1}^{k} H(|W_{i,.}^{(l)}| * |W_{.,i}^{(l)}| - \varepsilon)$$

Où H est la fonction de Heaviside, k la taille d'une sortie d'un layer et ε le seuil d'activation du neurone.
Pour sélectionner un modèle on cherchera alors à minimiser la somme des carrés de l'erreur de validation et de la *scarcity*.

3.2 *Transformer*

La seconde méthode consiste, comme décrit dans [3], à encoder nos expressions à l'aide d'un *Transformer*, dont le fonctionnement général est décrit en figure (1) .

2. Fraction de la variance expliquée.
3. Estimant le taux de solutions dont le plus grand écart à un point du jeu de données est inférieur à un certain seuil.

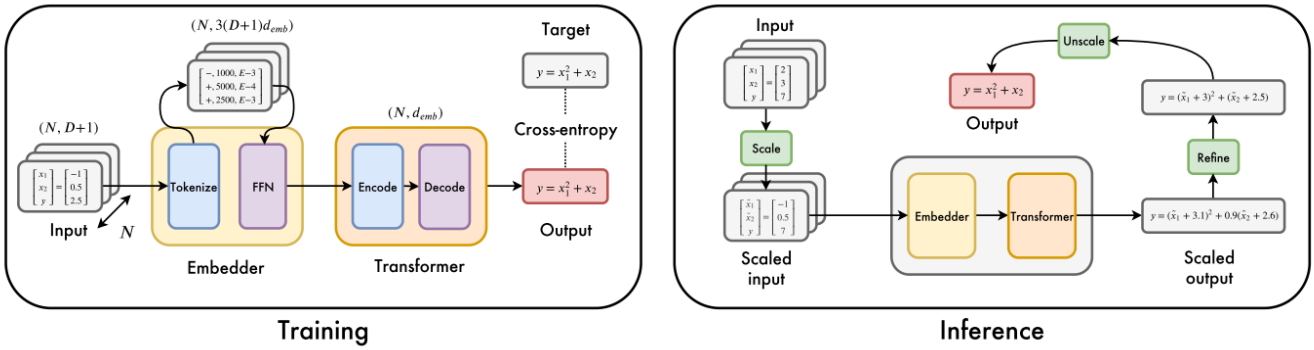


FIGURE 1 – Fonctionnement général du *Transformer*

3.2.1 Phase d'entraînement

Lors de la phase d'entraînement, chaque entrée $(x, f(x))$ est tout d'abord transformée en *token* : chaque nombre est représenté par son signe, sa mantisse et son exposant ainsi, par exemple, 2.37 est representé par $[+, 237, E-2]$. On fait ensuite passer ces *tokens* dans un FCNN pour les mettre à la dimension d_{emb}.
Ensuite, ces *tokens* sont données au *Transformer* constitué d'un encodeur et d'un décodeur dont le fonctionnement global est décrit par [5]. Cette partie permet de créer des *attention head* qui vont mettre en lumière les particularités de la fonction par rapport à son expression mathématique.
Enfin, on choisit nos paramètres de modèle en minimisant une *loss function* de type *cross-entropy* dont le taux d'apprentissage varie au cours des étapes. On finalise l'entraînement en utilisant un ensemble de validation.

3.2.2 Phase de validation

On pourra éprouver la qualité du modèle construit en mesurant (méthode du R^2 ou de la précision ϵ) sa capacité à prévoir correctement la fonction cible. On pourra l'évaluer dans différentes conditions :

- nombre d'opérateurs binaires, nombre d'opérateurs unaires
- dimension n de l'entrée
- nombre de points indicateurs $(\mathbf{x}, \mathbf{y})$.
- capacité d'extrapolation
- robustesse à l'application à y d'un bruit multiplicatif : $y_{\text{app}} = y(1 + \epsilon n)$, où $n \sim \mathcal{N}(0, \sigma)$.
- ...

3.2.3 Phase d'inférence

Lors de la phase d'inférence plusieurs éléments sont à bien prendre en compte.
Les variables utilisées pour l'entraînement du modèle sont normalisées (leur distribution est centrée et réduite), il doit donc en être de même pour les points pour lesquels on cherche l'expression mathématique. À l'étape de normalisation on en extrait les moyenne et variance, par lesquelles on transformera les variables de la fonction optimisée pour obtenir l'expression souhaitée. On pourra évidemment, par la suite, exploiter l'(éventuelle) capacité d'extrapolation (que l'on espère bien observer !), en lui passant des jeux de données d'une variance supérieure à 1.
De plus, une étape de raffinement est ajoutée à la sortie du *Transformer* afin de diminuer les erreurs dûes à la précision des constantes. Pour ce faire on optimise les paramètres à l'aide d'un algorithme de BFGS initialisé avec la prédiction du modèle à la sortie du *Transformer*.

Références

[1] LAMPLE, Guillaume et CHARTON, François. Deep learning for symbolic mathematics. arXiv preprint arXiv :1912.01412, 2019.

[2] S.-M. Udrescu, M. Tegmark, AI Feynman : A physics-inspired method for symbolic regression. Sci. Adv. 6, eaay2631 (2020).

[3] KAMIENNY, Pierre-Alexandre, D'ASCOLI, Stéphane, LAMPLE, Guillaume, et al. End-to-end symbolic regression with transformers. Advances in Neural Information Processing Systems, 2022, vol. 35, p. 10269-10281.

[4] Georg Martius Christoph H. Lampert. Extrapolation and learning equations. arXiv preprint arXiv :1610.02995, 2016

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is All You Need. arXiv preprint arXiv :1706.03762.