Département Sciences de la matière
École Normale Supérieure de Lyon
Université Claude Bernard Lyon I

Stage L3 2018-2019
Yohann Faure
L3 Physique

Réseau de neurones récursif pour la calibration en énergie des jets à CMS

La Physique des particules est l'étude des interactions et constituants élémentaires de la matière à l'échelle subatomique. À cette échelle, de nombreux paramètres et variables sont à prendre en compte, et de récentes méthodes basées sur le *Machine Learning* ont fait leur preuve comme moyen de calibration des grands détecteurs, tels que CMS (*Compact Muon Solenoid*). Ce stage porte sur le développement d'un programme basé sur un concept de *Machine Learning*, le réseau de neurones récursif, et sur son application à la correction en énergie des jets. Un tel développement est susceptible d'améliorer cette calibration fondamentalement nécessaire à la compréhension des phénomènes physiques étudiés au LHC.

Mots clefs : Physique des particules, CMS, Deep learning, RECNN.

Stage encadré par **Colin Bernet**
colin.bernet@cern.ch / tél. (+33) 4 72 44 80 65

Et par **Viola Sordini**
viola.sordini@cern.ch

Institut de Physique Nucléaire de Lyon
Domaine Scientifique de la Doua, Bâtiment Paul Dirac, 4 rue Enrico Fermi, 69622 Villeurbanne Cedex
https://www.ipnl.in2p3.fr/

16 novembre 2018

Remerciements

Je tiens à remercier Colin Bernet et Viola Sordini, mes maîtres de stage, qui m'ont proposé un sujet original et enrichissant, d'enjeu physique réel, et m'ont encadré pendant ces deux mois. J'adresse également un grand merci à Gaël Touquet, pour son encadrement et ses conseils précieux, qui a partagé mes galères au cours de ce stage et a travaillé avec moi à les résoudre. Je lui souhaite autant de courage dans la suite de sa thèse qu'il n'en a eu pendant ce stage.

J'adresse de plus des remerciements à toute l'équipe de l'IPNL, dont les membres ont toujours été accueillants et bienveillants envers moi. Échanger avec tant de personnes d'horizons si différents a été un réel plaisir et un enrichissement certain.

Enfin merci à mes relecteur·ices et ami·e·s, par ordre anti-alphabétique Youssef, Louis, Joël, Guillaume, Fanny et Aurore, qui ont donné de leur temps pour améliorer ce rapport, et à Lucas Torterotot, *maestro* du LaTeX, sans qui ces remerciements seraient probablement écrits en Comics Sans MS.

Table des matières

1 Introduction

Un jet est la manifestation expérimentale d'un quark ou d'un gluon de haute énergie. Produits lors des collisions de particules au LHC (*Large Hadron Colider*), ces quarks et gluons s'hadronisent en une flopée de particules d'énergie plus basse, qui sont ensuite mesurées par les grands détecteurs. Ce sont des phénomènes incontournables de la physique des particules moderne.

La calibration de l'énergie des jets du LHC est une tâche complexe nécessitant de nombreuses corrections. En effet l'information portée par un jet se compte en milliers de variables, toutes potentiellement décisives dans le calcul de l'énergie réelle de l'évènement mesuré.

Ce rapport présente la mise en place d'une méthode nouvelle de calibration de CMS (*Compact Muon Solenoid*), basée sur un article récent [1]. Cette méthode s'inspire de l'algorithme de reconstruction des jets et de la structure d'arbre qu'il révèle, et se base sur l'utilisation d'un réseau de neurones récursif (RECNN). L'objectif d'un tel développement est de réussir à calibrer l'énergie des jets.

Dans un premier temps, nous présenterons le contexte de cette étude, et le lien fort existant entre le problème physique étudié et les problématiques de programmation et de *Machine Learning*. Nous étudierons ensuite les algorithmes de reconstitution de jets utilisés dans CMS, afin de mettre en évidence la cohérence d'une approche récursive utilisant un RECNN. Enfin, nous étudierons plus en détail les choix de programmation et les résultats obtenus.

2 Le Contexte de l'étude

2.1 Les infrastructures de recherche

2.1.1 l'IPNL

L'Institut de Physique Nucléaire de Lyon (IPNL) est situé à Villeurbanne, sur le campus de la Doua. Il s'agit d'une Unité Mixte de Recherche, composée d'une centaine de chercheurs, enseignants-chercheurs et doctorants, ainsi que d'autant d'ingénieurs, techniciens et membres du personnel administratif. J'ai effectué mon stage sous la tutelle de Colin Bernet et Viola Sordini, chargés de recherche au CNRS.

2.1.2 Le CERN

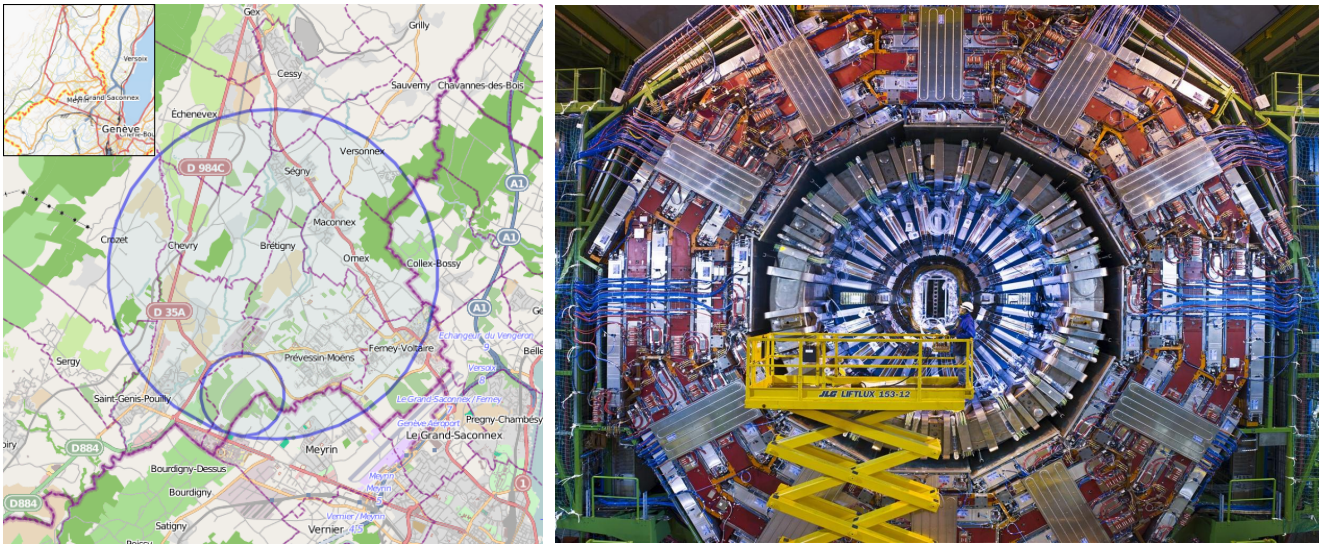
Fondé en 1954 sous l'appellation de Conseil Européen pour la Recherche Nucléaire, et maintenant appelé Organisation Européenne pour la Recherche Nucléaire, le CERN est à l'origine de nombreuses découvertes majeures en Physique Nucléaire et en Physique des Particules. Parmi les travaux les plus récompensés du CERN se trouvent la confirmation de l'existence des bosons W et Z (1983, Nobel 1984) et du boson de Higgs (2012, Nobel 2013).

Le CERN est aujourd'hui le plus grand centre de Physique des particules du monde. Situé à la frontière entre la France et la Suisse, il accueille notamment le LHC et le détecteur CMS, sur lequel porte ce stage.

2.1.3 Le LHC

Le LHC est un synchrotron de 27 km de circonférence, situé à une centaine de mètres sous terre. Ce dispositif expérimental colossal vise à accélérer des protons à très haute énergie et à les faire entrer en collision afin de vérifier expérimentalement des théories physiques telles que l'existence du boson de Higgs ou de la matière noire.

Les protons y sont accélérées par un champ électrique, et leur trajectoire suit la courbure de l'anneau grâce à un champ magnétique [2]. Deux faisceaux de protons circulent en sens opposés au sein de deux tubes sous vide. Chaque faisceau est constitué de paquets contenant un grand nombre de protons, accélérés à une énergie de 6.5 TeV (Tera électron-volt). Les faisceaux sont amenés à se rencontrer au centre des détecteurs du LHC, situés dans les cavernes expérimentales le long de l'anneau. Un croisement de paquet a lieu toutes les 25 ns, donnant lieu à une vingtaine de collisions proton-proton simultanées, à une énergie dans le centre de masse de 13 TeV.



(a) Localisation du LHC.

(b) Vue de la tranche de CMS.

Figure 1 – À gauche, une carte de la région de Genève. Le LHC y est représenté par un grand cercle bleu. À droite, le détecteur CMS vu de tranche.

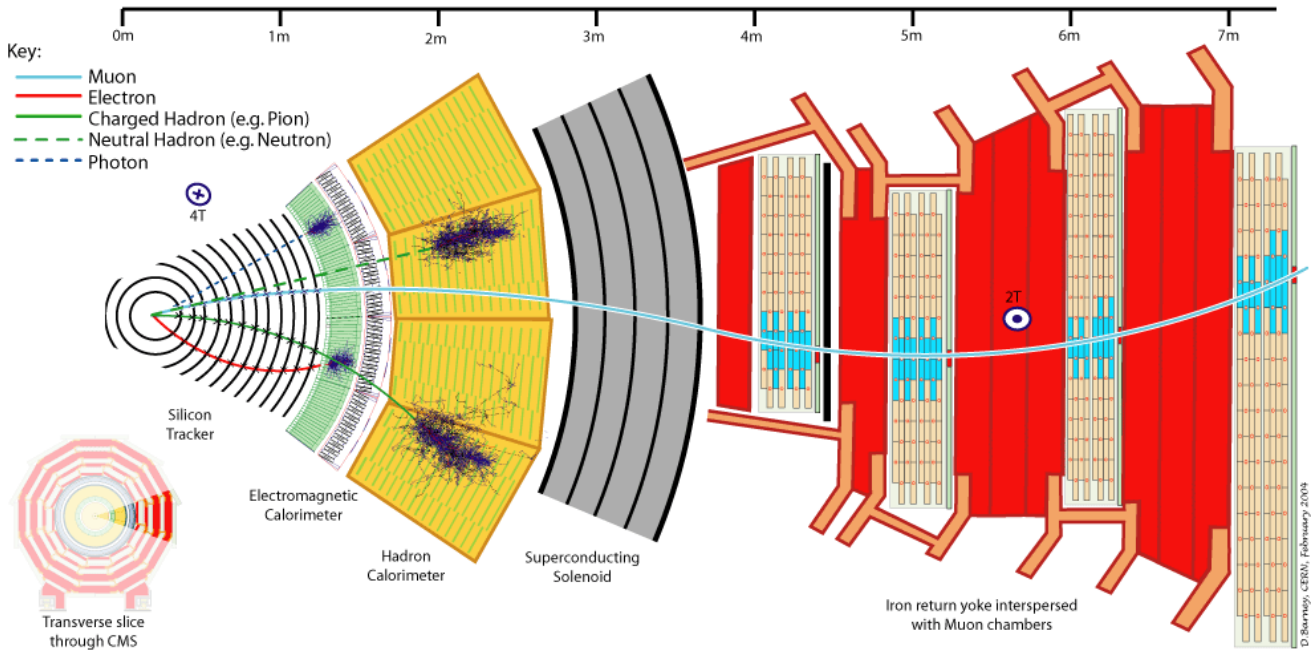


Figure 2 – Vue en coupe de la structure interne de CMS. Les lignes colorées partant du centre représentent des trajectoires typiques des différents types de particules dans le détecteur.

2.2 Pourquoi le *deep learning* ?

La réponse à cette question est double. Tout d'abord, le nombre de variables dont dépend la calibration que nous souhaitons effectuer est conséquent, de l'ordre de la centaine de particules par évènement, ayant chacune un quadrivecteur la caractérisant. Une telle structure donne à notre problème une grande proximité au traitement de l'image ou des langages. C'est d'ailleurs à l'aide de reconnaissance d'images que les premiers résultats à base de *deep learning* dans CMS ont été produits [3]. Nous nous focalisons ici sur un traitement plus proche de celui des langages. Ensuite, les essais déjà effectués sur des tâches similaires sont concluants [1].

2.3 Le matériel et les données utilisés

Les données que j'ai utilisées durant le stage sont des évènements simulés, produits à l'aide du logiciel Pythia [4]. Les conditions de simulation utilisées sont similaires aux conditions expérimentales réelles au LHC, le détecteur CMS étant simulé par le logiciel Geant4 [5].

J'ai principalement utilisé le langage de programmation Python 2.7 (plus d'information sur les modules utilisés en annexe), sur une station de travail dédiée munie d'un processeur à 20 threads.

2.4 Les conventions utilisées dans CMS

Dans cette section sont explicitées les variables et conventions que nous prendrons pour la suite de l'étude.

Si l'on approxime localement le tore du LHC par un cylindre, l'axe de symétrie du cylindre, qui est aussi l'axe sur lequel se déplacent les particules avant une collision, est nommé z. Le plan transverse à l'axe z, généré par les axes x et y, est nommé T. Les axes (x, y, z) forment une base orthonormée directe, y est vertical. Pour simplifier les calculs et les notations, nous prendrons par la suite la vitesse de la lumière $c = 1$. Avec ce choix, masse et énergie s'expriment dans la même unité (car $E = mc^2$), en GeV.

Les particules étudiées sont décrites par des quadrivecteurs impulsion-énergie, de la forme (p_x, p_y, p_z, E). Un rôle particulier est donné à l'impulsion transverse $p_T = \sqrt{p_x^2 + p_y^2}$. Cela correspond à l'impulsion dans le plan T. Celle-ci nous intéresse tout particulièrement du fait que l'impulsion initiale en z est inconnue.

3 La mesure des jets dans CMS

3.1 CMS

Le détecteur CMS [6], visible sur la Figure 1(a), est un détecteur polyvalent du LHC. De forme cylindrique, de 21 m de long et 15 m de diamètre, il est composé de plusieurs couches de détection, sensibles à différents types de particules, comme montré sur la Figure 2. L'élément principal est un solénoïde supraconducteur, qui produit un champ magnétique de 3.8 T en son centre, selon l'axe du cylindre. Ce champ magnétique permet de courber les trajectoires des particules chargées. La courbure renseigne sur la charge et l'impulsion des particules, et l'énergie déposée dans les calorimètres électromagnétique et hadronique permet de déterminer leur type.

3.2 La détection des particules

Lorsque des particules traversent les couches successives de CMS (Figure 2), elles interagissent différemment avec les sous-détecteurs selon leur nature. La combinaison des informations d'interaction permet de déterminer le type et l'énergie des particules de l'évènement.

Par exemple, lorsqu'une particule est détectée dans le calorimètre hadronique, c'est un hadron, dont la courbure de la trajectoire nous renseigne sur sa charge et son impulsion. Si une particule est détectée dans le calorimètre électromagnétique, c'est un photon ou un électron, les deux étant différenciés par leur trajectoire. Les muons quand à eux sont détectés dans les chambres à muons des couches externes de CMS.

Toutes les analyses de CMS sont basées sur l'algorithme de reconstruction du flux de particules («Particle Flow») [7], qui combine de manière optimale les informations collectées par chacun des sous-détecteurs pour identifier et reconstruire individuellement les particules de l'état final (hadrons, électrons, muons et photons).

3.3 Les corrections nécessaires

Dans un détecteur parfait, toutes les particules seraient détectées avec leur énergie exacte, cependant il y a des facteurs d'incertitude dans CMS.

Les détecteurs peuvent être inefficaces et ne pas détecter les particules, notamment lorsque leur énergie est faible. Ils ont de plus une résolution finie, empêchant une reconstruction exacte de l'énergie des particules produites dans la collision. Le *tracker*, utilisé pour la reconstruction des hadrons chargés a une excellente résolution, tout comme le ECAL, qui sert à obtenir l'énergie des photons. En revanche, la résolution du HCAL, qui sert à la reconstruction de l'énergie des hadrons neutres, est très mauvaise.

S'ajoute à cela le fait que l'erreur commise dépend fortement de la position de la particule à cause de la conception de CMS. Le détecteur est plus sensible en son centre que sur ses tranches.

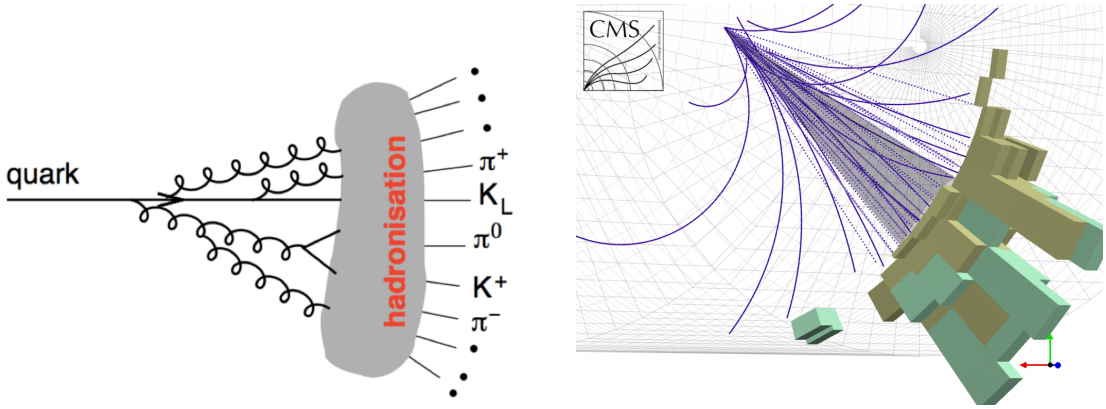


Figure 3 – Exemple de schéma d'hadronisation d'un quark en jet de particules (gauche), et visualisation d'un jet par CMS (droite).

3.4 La structure en jets

Lorsqu'une collision entre deux protons a lieu au centre de CMS, on parle d'évènement. Cette collision donne naissance à de nombreuses particules élémentaires. Les quarks et les gluons (particules à charge de couleur) ainsi engendrés s'hadronisent. Une particule en s'hadronisant donne naissance à un ensemble de particules regroupées dans un état final, comme montré sur la Figure 3. Ces ensembles de particules sont nommés des jets.

Les jets sont construits à partir des quadrivecteurs résultant de la reconstruction du flux de particules, par un algorithme de reconstruction de jets qui les regroupe suivant leur énergie et leur direction. Cet algorithme est détaillé dans la section 4. L'objectif de cet algorithme est de reconstruire des jets aussi ressemblants que possible aux jets physiquement générés afin de remonter au quadrivecteur du quark ou du gluon initial. Le logiciel utilisé dans CMS et dans notre étude pour effectuer cette opération est FastJet [8].

3.5 Reconstruction de l'énergie des jets

L'énergie brute du jet est obtenue en sommant les énergies des particules reconstruites utilisées pour construire le jet. Le détecteur étant imparfait, certaines particules, notamment à basse énergie, ne sont pas détectées. De plus, le *pileup* (superposition de plusieurs évènements) contamine les jets. L'énergie du jet brute n'est donc pas égale à l'énergie d'un jet qui aurait été reconstruit avec les particules vraies de l'état final (ou avec un détecteur parfait). Il faut donc corriger l'énergie des jets.

La méthode usuelle, dont le résultat est montré sur la Figure 4, est de se doter d'une série de corrections principalement fondées sur un traitement statistique de données de simulations. Afin d'évaluer la correction, on compare l'énergie trouvée à l'énergie du gen-jet. Celui-ci est calculé à l'aide des particules de sortie du logiciel de simulation Pythia, avant d'avoir simulé le détecteur.

Notons $p_T^{(gen)}$ le p_T du gen-jet, $p_T^{(rec)}$ la valeur reconstruite. La réponse du détecteur est définie comme la valeur $r = \frac{<p_T^{(rec)}>}{<p_T^{(gen)}>}$. Dans le cas d'un détecteur parfait, $r = 1$.

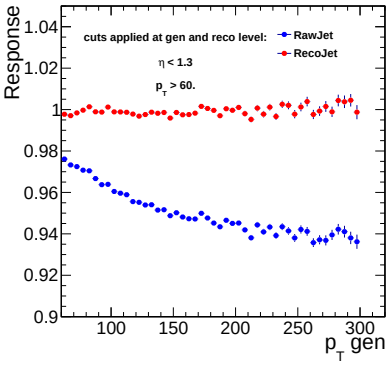


Figure 4 – Réponse en énergie des jets simulés en fonction de leur énergie réelle, en bleu sans correction, en rouge avec les corrections classiques de CMS.

4 L'algorithme de reconstruction

Afin de mieux comprendre le problème, j'ai été invité à implémenter moi-même un algorithme de reconstruction de jets. Cet outil, dont la cohérence avec FastJet a été vérifiée, est décrit ci-dessous.

4.1 Définitions

L'algorithme se base sur la définition de pseudojet, qui est une structure de données adaptée au problème. Il s'agit d'un ensemble de particules, muni d'un quadrivecteur défini comme la somme des quadrivecteurs des particules le composant. Les particules sont assimilées à des pseudojets contenant seulement une particule. Les pseudojets d'un évènement seront notés $\{J_i\}_{i\in\mathbb{N}}$.

Le but d'un tel algorithme est de passer d'un ensemble de N particules $\{P_i\}_{i\in[\![0,N-1]\!]}$ à un ensemble de M jets $\{J_i\}_{i\in[\![0,M-1]\!]}$. Pour ce faire, l'algorithme combine des pseudojets jusqu'à complétion d'une certaine condition d'arrêt.

Les valeurs associées au pseudojet i sont indicées par i. Pour rappel, si T est le plan transverse à l'axe z, $p_{Ti} = \sqrt{p_{xi}^2 + p_{yi}^2}$.

L'angle azimutal ϕ et l'angle polaire θ sont définis sur la figure 5. La rapidité y et la pseudorapidité η sont définies comme

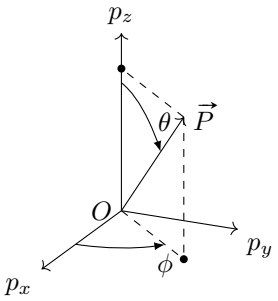


Figure 5 – Les coordonnées sphériques.

$$\eta = -\ln\tan\frac{\theta}{2}$$
$$y = \frac{1}{2}\ln\frac{E+p_z}{E-p_z}$$

On définit également ΔR_{ij}^2 et la (α,R)-distance entre les pseudojets i et j comme

$$\Delta R_{ij}^2 = (y_i - y_j)^2 + (\phi_i - \phi_j)^2$$
$$d_{ij}^{(\alpha,R)} = \min(p_{Ti}^{2\alpha}, p_{Tj}^{2\alpha})\frac{\Delta R_{ij}^2}{R^2}$$

La valeur de α correspond à différents comportements de la distance, différentes variantes de l'algorithme. Les valeurs généralement prises sont telles que $R \simeq 0.5$. Les variantes principales de l'algorithme sont kt ($\alpha = 1$), anti-kt ($\alpha = -1$), et Cambridge–Aachen ($\alpha = 0$).

4.2 L'algorithme

L'algorithme se déroule en plusieurs phases, décrites ci-dessous. Nous noterons d_{ij} la distance entre les pseudojets i et j sans spécifier R ou α.

`Initialisation :`

Pour chaque particule P_i, créer un pseudojet $J_i = \{P_i\}$.

`Exécution :`

$M := N$ (Nombre de particules dans l'évènement)
Créer la matrice de distance $D := (d_{ij})_{(i,j)\in[\![0,M-1]\!]}$
`Tant que` (Condition d'arrêt non vérifiée) **`:`**
    Chercher i et j tels que $d_{ij} = \min(D)$
    $J_i := J_i \cup J_j$
    Supprimer J_j et renuméroter
    $M := M - 1$ (Nombre de pseudojets)
    $D := (d_{ij})_{(i,j)\in[\![0,M-1]\!]}$

`Sortie :`

$\{J_i\}_{i\in[\![0,M-1]\!]}$

La condition d'arrêt dépend des variantes, mais les plus courantes sont des conditions sur le nombre de pseudojets voulu ou sur la distance maximale entre deux pseudojets à fusionner ; par exemple $\min(D) > d_{cut}$ où d_{cut} est une constante choisie par l'utilisateur. Selon les paramètres choisis, il y a une grande variation dans la forme des jets reconstruits, comme montré sur la Figure 6.

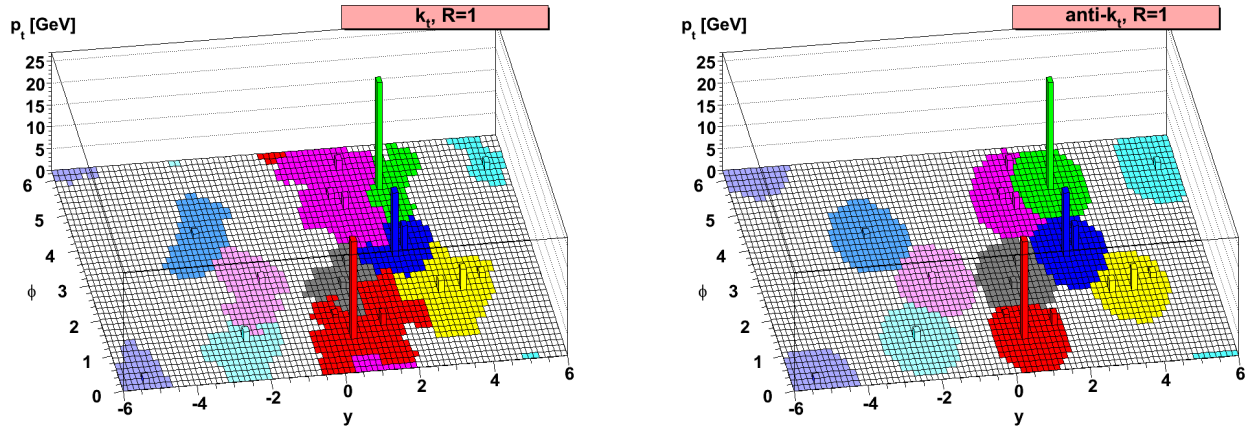


Figure 6 – Comparaison visuelle de la reconstruction effectuée par l'algorithme kt et l'algorithme anti-kt sur un évènement réel du LHC. Les deux algorithmes reconstruisent autant de jets, et les zones de haut p_T sont assemblées de manière similaire, mais les zones de bas p_T sont très variables [9].

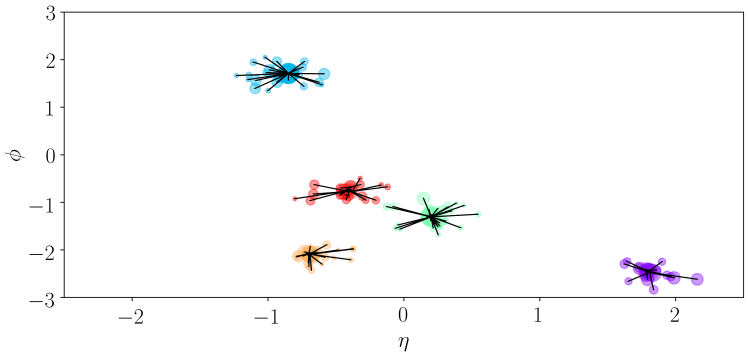


Figure 7 – Visualisation du fonctionnement de mon implémentation de l'algorithme de reconstruction. Un point correspond à une particule dans le plan (η,ϕ). Une couleur correspond à un jet final et la taille du point correspond au p_T de la particule.

Généralement, les particules sont représentées dans un espace à deux dimensions, (η,ϕ) ou (y,ϕ), comme sur la Figure 7, qui représente le regroupement en jets d'un évènement réel avec mon implémentation de l'algorithme anti-kt. Il y a dans cette distribution de particules une sous-structure, détectée par l'algorithme, qui divise les particules en cinq jets reconstruits. Pour plus de détails sur mon implémentation de l'algorithme usuel de reconstruction, voir le code en annexe de ce rapport.

4.3 La structure d'arbre engendrée

Pour obtenir un jet, l'algorithme regroupe séquentiellement les pseudojets qui le composent. Comme l'illustre la Figure 8, la construction d'un jet revient à la construction d'un arbre binaire. Un arbre est défini comme étant soit une feuille (une particule), soit un nœud (un pseudojet), contenant deux autres arbres, dits fils gauche et droit. Une écriture formelle d'une telle structure serait `arbre ::= arbre × arbre | feuille`.

De nombreux petits ajustements sur cette structure sont effectués, comme le passage d'information complémentaire du quadrivecteur, ou encore l'orientation des fils d'un nœud, mais ces détails sont explicités dans le code, disponible en annexe.

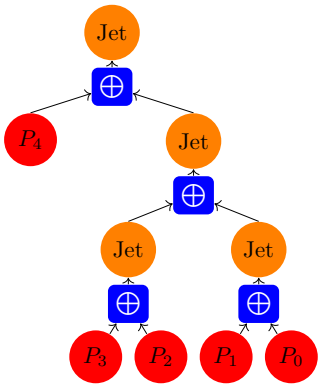


Figure 8 – Structure arborescente de l'algorithme de reconstruction d'un jet. Chaque nœud correspond à un pseudojet. En particulier, le dernier nœud correspond au jet final, et les feuilles aux particules initiales comprises dans le jet. Le symbole $\oplus$ représente la fusion de deux pseudojets.

5 Le *deep learning* pour la calibration

5.1 Fonctionnement d'un neurone

Un réseau de neurones est une structure de calcul inspirée pour sa forme sur le fonctionnement d'un réseau de neurones biologique. Un neurone est un outil de calcul qui prend des entrées scalaires et renvoie par une méthode de calcul déterministe une sortie scalaire, fonction des entrées [10]. Cette méthode repose sur un ensemble de poids, un biais, et une fonction d'activation. Ainsi un neurone à $n \in \mathbb{N}^*$ entrées $(x_1, \dots, x_n) \in \mathbb{R}^n$ doit être munis de n poids $(c_1, \dots, c_n) \in \mathbb{R}^n$, d'un biais $b \in \mathbb{R}$, et d'une fonction d'activation $f : \mathbb{R} \mapsto \mathbb{R}$. Sa sortie est alors $s = f(\sum_{i=1}^{n}(c_i x_i) - b)$. Cela donne en notant l'entrée comme un vecteur $\boldsymbol{X}$ et les poids comme un vecteur $\boldsymbol{C}$,

$$s = f(\boldsymbol{C}^{\mathrm{T}} \cdot \boldsymbol{X} - b)$$

L'intérêt d'une telle structure est la possibilité de faire évoluer les poids et le biais, afin d'effectuer une tâche dépendant des caractéristiques de chaque élément d'un jeu de données. Supposons par exemple que les entrées $X = (x_i)_i$ et la sortie désirée s_{gen} sont connues pour un jeu de données, dites données d'entraînement. Considérons une fonction dite fonction de coût $g : \mathbb{R}^2 \mapsto \mathbb{R}$ au moins $\mathcal{C}^1$ telle que $g(s(c_i, x_i, b), s_{gen})$ soit minimale lorsque $s = s_{gen}$. En minimisant la fonction g par rapport aux variables c_i et b pour toutes les données en même temps, il est possible de trouver des coefficients qui font converger la sortie du neurone vers la valeur désirée. Ce que l'on espère faire à partir de cela est de prédire le plus précisément possible s_{gen} d'un autre jeu de données, en supposant que le lien entre $\boldsymbol{X}$ et s_{gen} soit le même que pour les données d'entraînement.

Un exemple concret, mais trivial, serait celui d'un physicien ayant oublié le lien exact entre le carré de l'énergie d'une particule relativiste, le carré de sa masse et le carré de son impulsion, mais qui se souviendrait qu'un lien simple existe. S'il a des particules dont il connaît l'énergie, l'impulsion et la masse, il note $\boldsymbol{X} = (m^2, p^2)$ et $s_{gen} = E^2$. Il prend alors $f(x) = x$, $s = f(c_1 x_1 + c_2 x_2 - b)$ et $g(x_1, x_2, c_1, c_2, b, s_{gen}) = (s - s_{gen})^2$, et trouvera un minimum de g pour $(c_1, c_2, b) = (c^4, c^2, 0)$, et si on donne en entrée du neurone de nouvelles données, il prédira exactement la valeur du carré de l'énergie. Le physicien dans l'oubli n'aura qu'à lire les coefficients de $\boldsymbol{C}$ et b pour retrouver la fameuse relation $E^2 = m^2c^4 + p^2c^2$.

5.2 Fonctionnement d'un réseau de neurones

La principale limite d'un neurone seul est son incapacité à effectuer des tâches complexes. Le neurone est une structure très simple, qui ne devient réellement efficace que lorsqu'elle est assemblée en réseau. Pour fabriquer un réseau de neurones, il suffit de relier la sortie d'un neurone à l'entrée d'un autre neurone, comme montré dans le schéma de la Figure 9. Ce schéma montre une organisation en couche du réseau : la couche d'entrée (qui donne simplement les valeurs des variables) est densément connectée à la première couche (*i.e.* chaque entrée de la première couche est connectée à chaque sortie de la couche d'entrée), puis les sorties de la première couche forment une couche d'entrée pour la deuxième couche (et il serait possible de rajouter ainsi des couches). Le réseau est dit profond (*deep neural network*) lorsqu'il y a plus que trois couches intermédiaires entre les entrées et les sorties.

L'entraînement fonctionne de la même manière que pour un neurone seul, mais la fonction à minimiser est plus complexe et dépend des poids et biais de chacun des neurones. Les méthodes développées pour optimiser les coefficients sont nommées «*backpropagation*» [11]. Les exemples typiques de problèmes où les réseaux de neurones sont plus efficaces que des algorithmes classiques sont les problèmes à beaucoup de variables, et à sous-structures, comme les problèmes de détection de formes dans des images, ou le suivi et traitement de flux de données. Ils nécessitent néanmoins un échantillon d'entraînement conséquent.

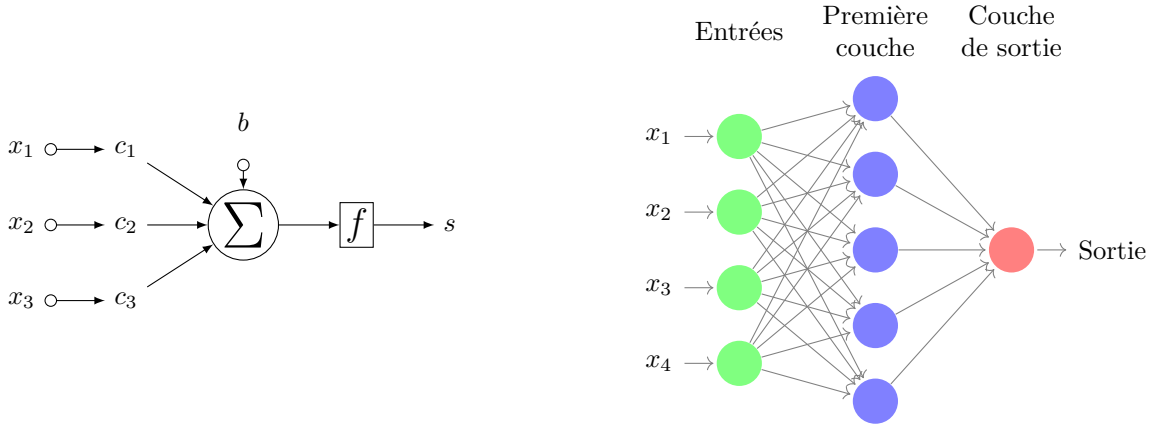


Figure 9 – Schéma du fonctionnement d'un neurone à gauche, d'un réseau à droite.

5.3 Vers le réseau de neurones récursif

Les réseaux de neurones simples, tel que celui montré sur la Figure 9, ont certaines limitations en terme de performances. Tout d'abord, plus un réseau est profond, plus la quantité de données nécessaires à l'entraînement est grande, et plus le calcul des coefficients est long (c'est le *vanishing gradient problem*).

Il est également à noter que pour certains problèmes spécifiques, la structure de données nécessite des transformations avant d'être injectée dans le réseau, ou même en milieu de réseau. C'est le cas des problèmes de traitement d'images, qui bénéficient grandement de la structure de réseaux dite convolutionnelle [12]. L'approche des réseaux convolutionnels a déjà été mise en place pour la gestion sous forme d'images des jets dans CMS [3], cependant cette méthode implique une perte d'information lors de la pixellisation des particules du jet.

Une méthode par réseau de neurones classique nécessite de plus un nombre fixe de variables en entrées, or dans notre cas, les jets peuvent posséder un nombre de particules très variable, allant de une à plusieurs centaines. Nous avons tenté de contourner ce problème en sélectionnant un nombre fixé de variables bien supérieur au nombre de variables dans un jet, et de remplir par la valeur -99 les variables qui n'avaient pas de correspondance physique. Le réseau voyait donc d'abord les particules réelles de l'évènement, puis un certains nombre de particules contenant uniquement la valeur -99. Les résultats étant très mitigés, nous avons décidé d'utiliser une structure plus adaptée et plus complexe, le réseau de neurones récursif (ou RECNN, pour *RECursive Neural Network*).

5.4 Principes du RECNN

La structure d'un jet étant naturellement arborescente, comme montré en section 4.3, l'implémentation d'un réseau basé sur cette structure semble particulièrement adaptée au problème. Le principe du RECNN est d'appliquer le même réseau de manière récursive à tous les nœuds de l'arborescence, en prenant comme variables d'entrée non seulement les informations du nœud considéré, mais aussi la sortie du réseau appliqué aux fils du nœud. Ce processus est détaillé sur la Figure 15 en annexe.

Plus formellement, définissons cette application comme une fonction F_{RECNN}, qui prend en entrée le vecteur des variables du nœud et les vecteurs des variables de sortie des fils.

Pour une feuille, on applique $F_{RECNN}((x_1, \dots, x_n), \mathbf{0}, \mathbf{0})$, où $(x_i)_i$ sont les variables de la particule associée à la feuille.

Pour un nœud, on applique récursivement $F_{RECNN}((x_1, \dots, x_n), (s_{FG}), (s_{FD}))$, où s_{FG} (respectivement s_{FD}) est la sortie du réseau, appliqué au fils gauche (respectivement droit).

Une fois cette étape terminée, il suffit d'appliquer un dernier réseau de neurones classique sur les sorties du RECNN à la racine de l'arbre.

5.5 Les défauts du RECNN

L'entraînement d'un réseau est une succession de calculs matriciels simples, en grande quantité. Ainsi, une amélioration classique de l'implémentation d'un réseau de neurones est d'effectuer les calculs en parallèle sur un GPU. Les cartes graphiques ayant de nombreux cœurs, un implémentation sur GPU accélère beaucoup l'entraînement. Les modules adaptés pour faire cela en python sont Keras [13] et TensorFlow [14].

Cependant, le premier et majeur défaut du RECNN, qui m'est apparu très rapidement, est que le nombre faible de calculs effectués simultanément et la structure du code ne permettent pas de porter efficacement le programme sur GPU. Ainsi l'étude est limitée par la puissance de calcul du CPU.

Le deuxième défaut est l'exoticité d'une telle implémentation. En effet, les implémentations de réseaux récursifs sont rares, et le support de ces derniers par les modules usuels de *deep learning* comme Keras est inexistant. J'ai dû réimplémenter l'intégralité du fonctionnement du réseau sans ces outils pourtant très aboutis et efficaces.

6 L'implémentation du RECNN

Le travail principal de ce stage a été l'implémentation d'un RECNN adapté à notre problème, en s'inspirant et en améliorant un code déjà existant [1]. Cette section décrit cette implémentation.

6.1 La compréhension du code

L'équipe et moi avons décidé de nous baser sur le code associé à l'article de Louppe et coll. [1]. Pour ce faire, nous avons contacté l'auteur afin qu'il nous fournisse ses données d'entrée et son code. La compréhension du format

d'entrée était nécessaire à la compréhension du programme, du fait du manque de commentaires dans ce dernier. La lecture du code et sa compréhension détaillée ont pris plus d'une semaine, mais à la fin de ce travail j'ai pu schématiser les étapes de fonctionnement de celui-ci, et envisager les adaptations nécessaires.

6.2 Les structures de données utilisées

Les structures utilisées pour les jets sont principalement des tableaux Numpy et des dictionnaires.

En entrée du processus, les jets reconstruits par CMS sont stockés sous forme de *Tree* dans le logiciel Root. Gaël Touquet m'a fourni un code python transformant ce *Tree* en tableau contenant quadrivecteur et le type de chacune des particules, regroupées en jet, et la valeur de sortie désirée, $p_T^{(gen)}$.

La structure utilisée en entrée du réseau de neurones est une structure de dictionnaire, contenant une description de l'arborescence reconstruite sous forme de tableau, la liste des quadrivecteurs des particules, ainsi que des informations telles que η ou p_T du jet.

6.3 Optimisations

Les changements de structure majeurs effectués très tôt dans la ré-implémentation du code ont été de remplacer le module Pickle par Numpy partout où une sauvegarde sur disque était effectuée. En effet, bien que Pickle soit un utilitaire simple d'utilisation et relativement efficace, Numpy s'est montré largement meilleur, comme l'indique la Figure 10. On y observe un gain d'un facteur 25.

J'ai également proposé une implémentation de multiprocesseur pour les fonctions utilisées. La machine sur laquelle l'équipe travaille possédant 20 cœurs, la différence de performance a été radicale.

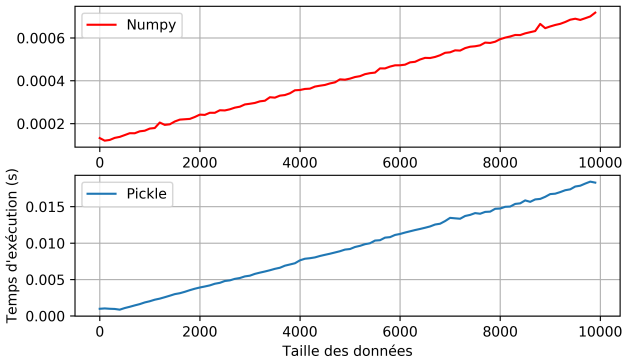


Figure 10 – Comparaison de l'efficacité de Numpy et de Pickle. La taille des données est linéaire, en unité arbitraire.

6.4 Le pré-traitement

6.4.1 Présentation

La première étape du programme est d'effectuer un pré-traitement des données avant l'entrée dans le réseau. Ce pré-traitement consiste en une réorientation des jets afin de leur donner une structure similaire aux yeux du RECNN. Ce pré-traitement comporte plusieurs étapes. Tout d'abord, il effectue une re-reconstruction du jet, afin de déterminer son arborescence. Celle-ci peut être effectué en kt, anti-kt ou Cambridge-Aachen, mais également de manière aléatoire, ou en ordonnant les particules en p_T.

Les fonctions de pré-traitement effectuent ensuite un boost du jet de $-p_z$. Cela correspond à une transformation de Lorentz sur les particules qui projette le quadrivecteur du jet dans le plan transverse. L'étape suivante est une rotation de l'ensemble des particules. Cette rotation s'effectue autour d'une sous-structure du jet. Selon les paramètres choisis, le jet est subdivisé en sous-jets à l'aide de l'algorithme de reconstruction. La rotation se fait autour du sous-jet de plus haut p_T, et oriente le deuxième sous-jet tel que $\eta = 0$. Le p_T du jet est invariant par ces transformations.

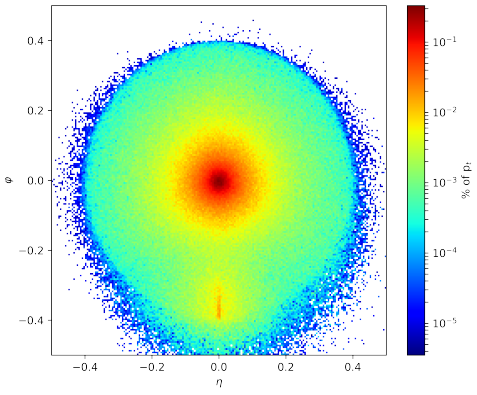


Figure 11 – Visualisation de l'ensemble des jets après le pré-traitement en utilisant l'algorithme kt.

Le résultat du pré-traitement est montré sur les Figures 11 et 12. Chaque case se voit attribuer une couleur selon le pourcentage de l'énergie totale qu'elle contient, pour l'ensemble des jets des données d'entraînement. Il y a effectivement une concentration d'énergie dans la région centrale, qui vient directement de la structure interne des jets, et une petite sous-structure suivant l'axe vertical, due à la rotation.

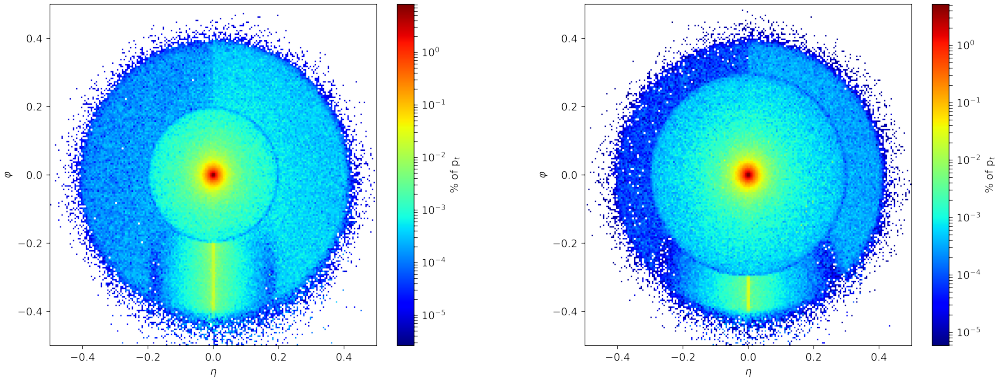


Figure 12 – Résultat du pré-traitement en utilisant l'algorithme anti-kt, avec $R = 0.2$ puis $R = 0.3$.

6.4.2 Complexité et détails d'implémentation

La fonction est de complexité linéaire avec le nombre de jets en entrée. La majeure partie du code était prévue pour fonctionner sur un seul cœur de CPU, de manière séquentielle. Pourtant, chaque jet est indépendant des autres, le pré-traitement peut être effectué en parallèle sur tous les cœurs du CPU. J'ai donc mis en place un traitement parallèle pour le pré-traitement, gagnant ainsi un facteur 15 dans son exécution.

Cette amélioration cumulée au passage en Numpy a permis de passer d'un temps d'exécution de plus d'une heure pour un jeu de paramètres et de données à un pré-traitement de moins de deux minutes. Grâce à cela il a été possible de faire de nombreux essais successifs pour le débogage du programme et pour l'optimisation des paramètres de pré-traitement.

6.5 L'entraînement

6.5.1 Le fonctionnement général

Une fois le pré-traitement effectué, les données sont prêtes pour l'entraînement du réseau. La fonction de coût que nous avons choisie est un écart quadratique de la valeur prédite à la vraie valeur, $g(p_T^{(raw)}, p_T^{(gen)}) = (p_T^{(raw)} - p_T^{(gen)})^2$. Cette fonction voit sa valeur diminuer au cours de l'entraînement, et finit par stagner lorsque le réseau atteint un optimum local.

L'entraînement s'effectue sur un jeu de données fini. Dans l'idéal le jeu de données doit être le plus grand possible, mais pour palier à des problématiques d'espace disque, l'usage est d'entraîner le réseau sur le même jeu de données plusieurs fois. Chaque passage du jeu de données s'appelle une époque.

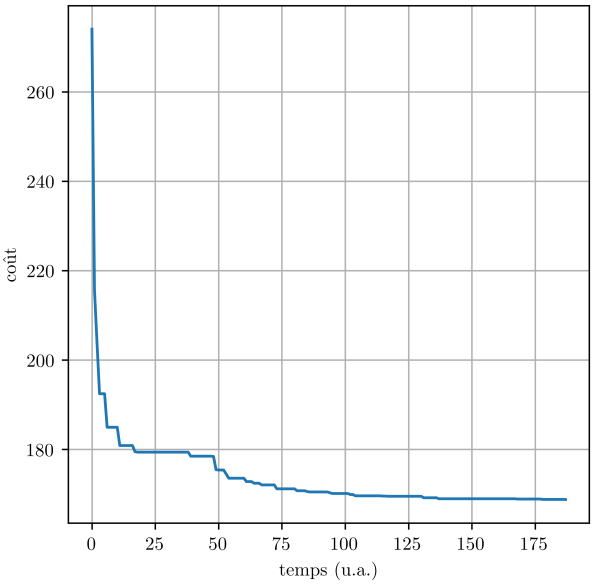


Figure 13 – Évolution de la fonction de coût au cours de l'entraînement.

6.5.2 Les détails d'implémentation

De la même manière que le pré-traitement était optimisé pour un seul cœur, l'entraînement n'avait pas été entièrement pensé pour un traitement parallèle. J'ai donc amélioré la distribution multi-processeur pour les fonctions d'entraînement. J'ai également décidé d'utiliser telle quelle une partie du code du RECNN.

Les options d'entraînement sont nombreuses, des paramètres du réseau aux paramètres de backpropagation. Afin d'orienter le réseau vers un fonctionnement efficace, il faut également sélectionner les informations qu'il prend en entrée. Pour plus de détails sur ces points, consulter la documentation du code sur GitHub. Un schéma plus détaillé du fonctionnement du RECNN est disponible sur la Figure 15 en annexe.

7 Résultats de calibration

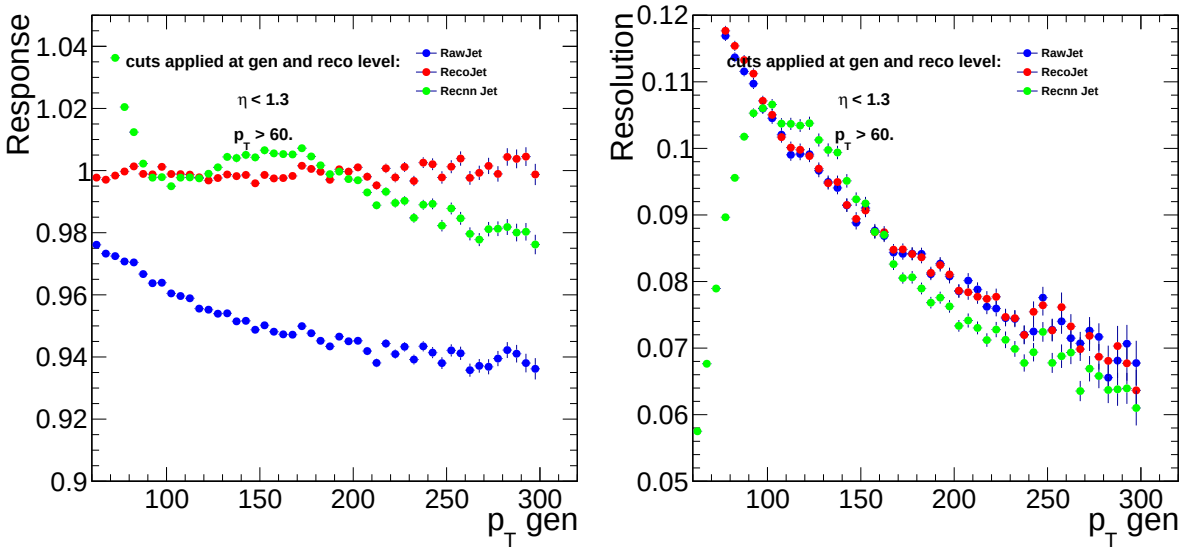


Figure 14 – Résultats de la calibration à l'aide du RECNN. Réponse (gauche) et résolution relatives (droite) sur le p_T du jet pour le jet brut (RawJet), le jet corrigé par CMS (RecoJet), et le jet corrigé par le RECNN (RecnnJet). Un détecteur parfait donnerait la constante 1 pour la réponse et la constante 0 pour l'écart relatif.

Le processus complet de pré-traitement et d'entraînement dure environ 3 heures pour un échantillon d'entraînement de 300 Mo avec les paramètres par défaut du programme sur une machine à 20 cœurs. La complexité de l'ensemble de l'algorithme est linéaire en le nombre de jets dans la donnée d'entraînement. L'évaluation du réseau est également linéaire et prend pour le même type de données 5 minutes environ.

La méthode d'évaluation choisie pour la régression avec le RECNN est la même que pour la méthode classique, et la réponse du RECNN peut être vue sur la Figure 14.

L'espace des $p_T^{(gen)}$ a été découpé en tranches de 5 GeV. Pour chaque tranche, nous avons ajusté l'histogramme de $\frac{p_T^{(rec)}}{p_T^{(gen)}}$ par une courbe gaussienne. Nous avons ensuite tracé la moyenne et l'écart type divisé par la moyenne de l'ajustement gaussien en fonction de $p_T^{(gen)}$.

La réponse calibrée par le RECNN est bien meilleure que la réponse non calibrée à $p_T > 100$, mais moins bonne que la calibration classique. Il y a notamment une sur-estimation à bas p_T et une sous-estimation à haut p_T.

La chute à haut p_T est probablement due à un manque de statistique pour l'entraînement, la distribution en p_T de l'échantillon étant décroissante. La même explication est possible pour la surestimation à bas p_T, car les données d'entraînement étaient coupées à $p_T^{(gen)} = 60$, ce qui rend l'ajustement gaussien invalide. Ce type d'effet de bords est à éliminer, en s'interdisant par exemple de calibrer les jets tels que le p_T mesuré sans correction $p_T^{(raw)} < 100$.

Il est également remarquable que l'écart type de la méthode RECNN est plus faible que pour les deux autres méthodes. Il est possible que cela indique que le réseau a réussi à extraire des informations pertinentes des données d'entraînement. Cela reste tout de même mitigé en raison de la faible quantité de données dans les gammes de p_T considérées.

8 Conclusions et perspectives

8.1 Conclusions de l'étude

Les résultats que j'ai obtenus sont une amélioration par rapport à l'utilisation de données non calibrées, mais la calibration standard de CMS reste la meilleure. Cependant, cette étude est un premier pas vers une nouvelle méthode de calibration de l'énergie des jets dans CMS. En effet en implémentant cet algorithme, nous avons montré qu'une méthode de calibration par *deep learning* est possible.

L'algorithme que j'ai implémenté est fonctionnel, et peut être appliqué à d'autres jeux de données. Le code est de plus polyvalent, et peut être utilisé pour d'autres usages, notamment pour la discrimination (étiquetage de la saveur du parton ayant initié le jet, ou discrimination des τ).

La grande limitation aujourd'hui repose à mon sens sur le temps, le temps d'entraînement tout d'abord, et le temps de développement ensuite. Je pense en effet qu'une plus longue durée de stage aurait pu permettre un développement plus abouti de la technique. Parmi les améliorations prévues par Gaël Touquet pour la suite du développement se trouve l'optimisation des paramètres du réseau final, notamment du nombre de couches et des fonctions d'activations choisies.

8.2 Conclusions personnelles

Ce stage m'a permis de découvrir réellement le monde de la recherche, ses hauts et ses bas, et ses difficultés. J'ai pu ressentir le plaisir de trouver des résultats après plusieurs heures de travail et de calcul intensif, et la déception de voir mon programme planter après un temps équivalent. De toutes ces expériences, celle qui m'a le plus appris est la communication entre chercheurs. Parler de mon travail et entendre parler de celui des autres m'a permis de découvrir d'autres modes de pensée, d'autres modes d'expression, et d'apprendre à faire preuve de pédagogie lors de l'explication de concepts complexes.

Références

[1] G. Louppe et coll. « QCD-Aware Recursive Neural Networks for Jet Physics » (2017). arXiv : 1702.00748 [hep-ph].

[2] E. M. McMillan. « The Synchrotron—A Proposed High Energy Particle Accelerator ». *Phys. Rev.* 68 (5-6 1945), p. 143-144. doi : 10.1103/PhysRev.68.143. url : https://link.aps.org/doi/10.1103/PhysRev.68.143.

[3] J. Cogan et coll. « Jet-images : computer vision inspired techniques for jet tagging ». *Journal of High Energy Physics* 2015.2 (2015), p. 118. doi : 10.1007/JHEP02(2015)118. url : https://doi.org/10.1007/JHEP02(2015)118.

[4] T. Sjöstrand, S. Mrenna et P. Skands. « PYTHIA 6.4 physics and manual ». *Journal of High Energy Physics* 2006.05 (2006), p. 026. url : http://stacks.iop.org/1126-6708/2006/i=05/a=026.

[5] S. Agostinelli et coll. « GEANT4 : A Simulation toolkit ». *Nucl. Instrum. Meth.* A506 (2003), p. 250-303. doi : 10.1016/S0168-9002(03)01368-8.

[6] The CMS Collaboration. « The CMS experiment at the CERN LHC ». *Journal of Instrumentation* 3.08 (2008), S08004. url : http://stacks.iop.org/1748-0221/3/i=08/a=S08004.

[7] A. M. Sirunyan et coll. « Particle-flow reconstruction and global event description with the CMS detector ». *JINST* 12.10 (2017), P10003. doi : 10.1088/1748-0221/12/10/P10003. arXiv : 1706.04965 [physics.ins-det].

[8] M. Cacciari, G. P. Salam et G. Soyez. « FastJet User Manual ». *Eur. Phys. J.* C72 (2012), p. 1896. doi : 10.1140/epjc/s10052-012-1896-2. arXiv : 1111.6097 [hep-ph].

[9] M. Cacciari, G. P. Salam et G. Soyez. « The Anti-k(t) jet clustering algorithm ». *JHEP* 04 (2008), p. 063. doi : 10.1088/1126-6708/2008/04/063. arXiv : 0802.1189 [hep-ph].

[10] F. Rosenblatt. « The Perceptron : A Probabilistic Model for Information Storage and Organization in The Brain ». *Psychological Review* (1958), p. 65-386.

[11] D. E. Rumelhart, G. E. Hinton et R. J. Williams. « Learning representations by back-propagating errors ». *Nature* 323 (oct. 1986), p. 533-536. doi : 10.1038/323533a0.

[12] Y. Lecun et Y. Bengio. « Convolutional networks for images, speech, and time-series ». English (US). *The handbook of brain theory and neural networks*. Sous la dir. de M. Arbib. MIT Press, 1995.

[13] F. Chollet et coll. *Keras*. https://keras.io. 2015.

[14] M. Abadi et coll. « TensorFlow : A system for large-scale machine learning ». *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, p. 265-283. url : https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

Annexes

Structure détaillée du RECNN

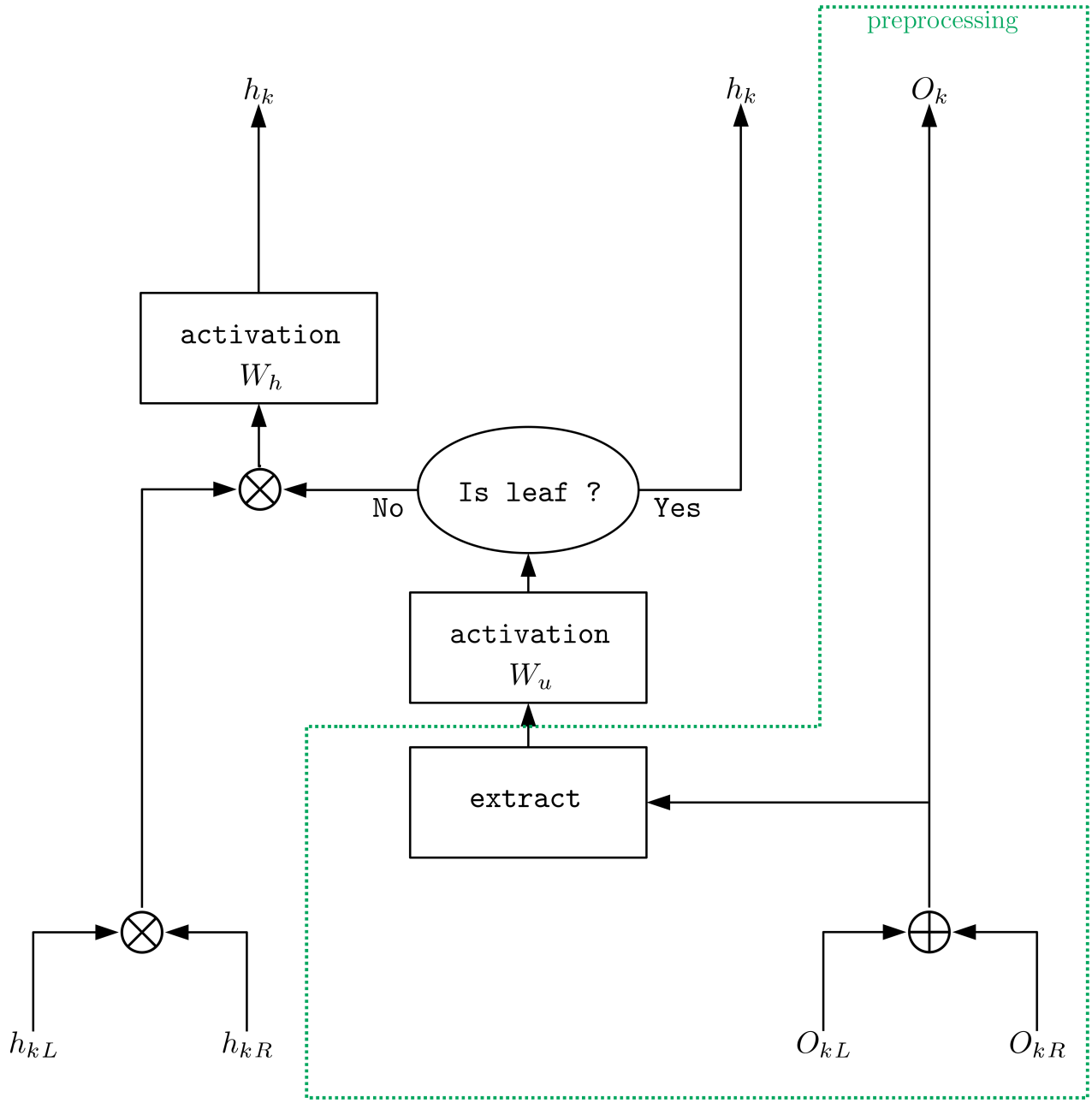


Figure 15 – La structure du réseau RECNN.

À chaque nœud du réseau sont calculées deux valeurs, O_k et h_k, qui correspondent respectivement au quadrivecteur du nœud et à la sortie du RECNN appliqué au nœud. Les boîtes W sont les parties du réseau qui sont entraînées. La boite verte correspond à la partie de la tâche effectuée pendant le pré-traitement des données. La boite `extract` correspond à l'extraction des paramètres donnés en entrée du RECNN. Le symbole $\oplus$ correspond à une somme terme à terme des quadrivecteurs, et $\otimes$ à une concaténation.

Il est à noter que la structure décrite ici est celle du RECNN simple. Nous avons suivi la démarche de notre article de référence [1] et implémenté une version plus complexe du réseau, dite *gated*. Cette amélioration complexifie l'entraînement mais améliore les performances en donnant une forme de mémoire et de possibilité d'oubli au réseau.

Lien GitHub

Ceci est un lien vers mon GitHub. Il contient les codes que j'ai produit, ainsi qu'une documentation succincte sur le fonctionnement du RECNN. Étant donné que le code continue d'évoluer et que je quitte l'équipe de développement, voici un lien vers le GitHub de Gaël Touquet, qui héberge maintenant tous les derniers changements.

La discrimination

Le programme que j'ai écrit est destiné à être modulaire, et peut tout à fait servir à la discrimination. La documentation explique les options à sélectionner pour effectuer une discrimination entre deux types de jets.

Afin de tester le programme, nous avons effectué une discrimination jet-τ. Lorsqu'un τ est produit dans CMS, il peut s'hadroniser, comme un quark ou un gluon. Cependant, différencier ceux-ci est primordial, par exemple pour la recherche sur le boson de Higgs. Ce boson peut se décomposer en deux τ. Pour remonter jusqu'aux données concernant le Higgs, il faut donc être capable de trouver les évènement contenant deux τ.

En observant la répartition en énergie après pré-traitement, sur la Figure 16, il est visible que les τ sont nettement plus condensés autour de la particule de plus haut p_T. Cette distinction doit être apprise par le réseau de neurones.

Nous n'avons pas eu le temps d'obtenir des résultats pour cette étude, mais la suite du développement se focalisera probablement dessus.

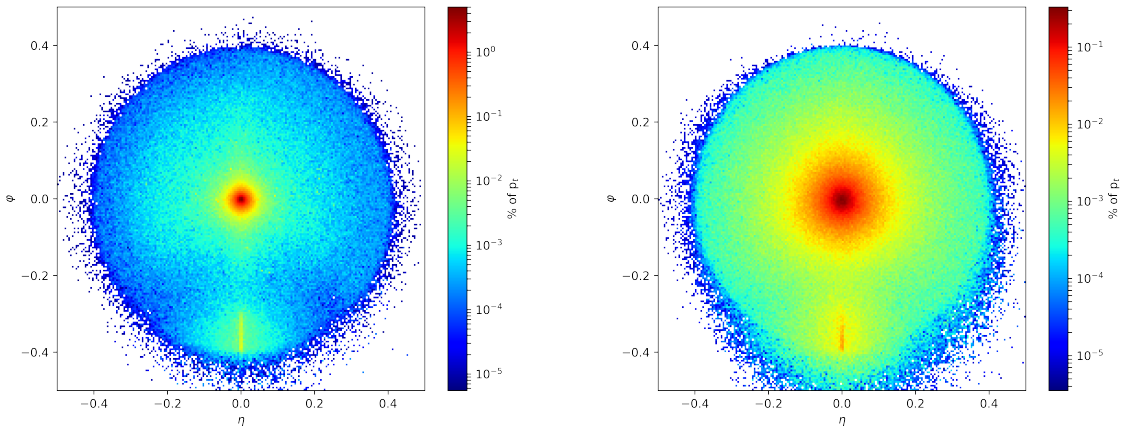


Figure 16 – À gauche la répartition de l'énergie des τ hadroniques, à droite celle des jets.

Code de l'algorithme de reconstruction

```
"""
This program takes as input a file containing :
[(jet 1]
 (jet 2)
 ...
]

with (jet1)=(jet_e,jet_px,jet_py,jet_pz,jet_q,jet_m,
             genjet_e,...
             particle0_e,...
             ...)
"""



import numpy as np
from scipy import spatial
import sys
import matplotlib.pyplot as plt
import matplotlib.cm as cm


##### DATA LOADING AND SHAPING #####


if len(sys.argv)==1:
        data = np.load('/home/yohann/Desktop/stage/data/sample_QCD_yohann_1event.npy')
else :
        data = np.load('sys.argv[1]')

#saving types
datatypes = eval(str(data.dtype))
datatypes = np.array([i[0] for i in datatypes])

#going from array of tuples to bi-dimensional array
data = np.array([list(i) for i in data])

#selecting usefull columns
selectedcol = np.concatenate([[i+1,i+2,i+3,i] for i in range(12,data.shape[1]) if i%6 == 0])
finaljets = data[:,[1,2,3,0]]
data=data[:,selectedcol]
datatypes = datatypes[selectedcol]

#still reshaping
databis = np.reshape(data,(data.shape[0],data.shape[1]//4,4))

##### DEFINGING CLASS AND FUNCTIONS #####
```

```
def count(timing):
        """
        just a counter so i know whitch jet formed first
        """
        def wrapper(*args, **kwargs):
                wrapper.counter += 1    # executed every time the wrapped function is called
                return timing(*args, **kwargs)
        wrapper.counter = 0         # executed only once in decorator definition time
        return(wrapper)

@count
def timing():
        """
        counting function
        """
        pass


class Jet():
        """
        class to represent a particle jet
        If the jet is the i^th particle, Jet(p4_of_i,particle=i)
        If the jet is a composed jet, composed by jeta and jetb,
    Jet(p4_of_the_sum, left=jeta,right=jetb)
        """
        def __init__(self, p4init, left=None, right=None, particle = -1):
                #self physical data
                self.p4       = p4init
                self.y        = 0.5 * np.log( (p4init[3] + p4init[2]) /
                              (p4init[3] - p4init[2]) )
                self.phi      = np.arccos( p4init[0] / np.sqrt( p4init[0] * p4init[0] +
                              p4init[1] * p4init[1] ) ) * np.sign(self.p4[1])
                self.theta    = np.arccos( p4init[2] / np.sqrt( p4init[2] ** 2 +
                              p4init[1] ** 2 + p4init[0] ** 2) )
                self.eta      = -np.log(np.tan(self.theta/2))
                self.energy   = p4init[-1]
                self.pt       = np.sqrt( p4init[0] * p4init[0] + p4init[1] * p4init[1] )
                #self composition, as the fusion of two other jets
                self.left     = left
                self.right    = right
                self.particle = particle
                timing()
                self.date_of_creation = timing.counter
                return(None)
        #Useful functions for the jets

        def particles_list(self):
                """
```

```
            return a list of all particles in the jet
            """
            if self.particle != -1 :
                    return([self.particle])
            return(sorted(Jet.particles_list(self.left)+Jet.particles_list(self.right)))

    def __add__(self,other):
            """
            addition between jets
            """
            return(Jet(self.p4+other.p4,left=self,right=other))

    def plot(self, color=None):
            """
            a clean function to plot the jets and particles
            """
            if self.particle == -1 :
                    plt.arrow(self.left.eta,self.left.phi, self.eta - self.left.eta,
                                self.phi - self.left.phi, head_width=0.01, head_length=0.01)
                    plt.arrow(self.right.eta, self.right.phi, self.eta - self.right.eta,
                                self.phi - self.right.phi,head_width=0.01, head_length=0.01)
                    if self in to_do_jets_list :
                            plt.scatter(self.eta,self.phi,color=cm.rainbow(
                                        np.linspace(0,1,len(to_do_jets_list)))
                                        [belongs_to_final_jet_number(self)],
                                        s=50*np.log(self.energy+1),alpha=0.4)
                    else :
                            plt.scatter(self.eta,self.phi,color=cm.rainbow(
                                        np.linspace(0,1,len(to_do_jets_list)))
                                        [belongs_to_final_jet_number(self)],
                                        s=50*np.log(self.energy+1),alpha=0.05)
            else :
                    plt.scatter(self.eta,self.phi,color=cm.rainbow(
                                np.linspace(0,1,len(to_do_jets_list)))
                                [belongs_to_final_jet_number(self)],
                                s=50*np.log(self.energy+1),alpha=0.4)
            return(None)

    def particle_history(self):
            if self.particle != -1:
                    return(self.particle)
            else :
                    return([self.left.particle_history(),self.right.particle_history()])

def distance(p4a, p4b):
        """
        Compute the distance between 2 particles with p4 p4a and p4b.
        Ex : p4a = np.array([px,py,pz,E]).
```

```
        """
        global alpha
        jeta,jetb=Jet(p4a),Jet(p4b)
        if (jeta.p4 == jetb.p4).all():
                return(sys.float_info.max)
        R2 = 0.4**2
        deltaphi = (jeta.phi - jetb.phi) % (2*np.pi)
        if deltaphi > np.pi:
                deltaphi += -2*np.pi
        DeltaR2 = (jeta.y-jetb.y) ** 2 + ( deltaphi ) ** 2
        c = min( jeta.pt ** alpha, jetb.pt ** alpha ) * DeltaR2 / R2
        return(c)


def combine(i,j):
        """
        Allows you to combine two jets in the to_do_jets_list,
        at index i and j in said list, and updates the distance matrix.
        """
        global distance_matrix
        if i>j:
                i,j=j,i
        done_jets_list.append(to_do_jets_list[i])
        to_do_jets_list[i] = to_do_jets_list[i] + to_do_jets_list[j]
        done_jets_list.append(to_do_jets_list.pop(j))
        distance_matrix=np.delete(distance_matrix,j,0)
        distance_matrix=np.delete(distance_matrix,j,1)
        for k in range(len(distance_matrix)):
                distance_matrix[i][k]=distance_matrix[k][i]=
                        distance(to_do_jets_list[i].p4,to_do_jets_list[k].p4)
        return(None)


def belongs_to_final_jet_number(jet):
        """
        allows you to know in which jet the jet you test is included in the end
        """
        particles = jet.particles_list()
        for i,f in enumerate(final_jets_particles):
                if set(f) & set(particles):
                        return(i)
        return(None)

def plot_jets():
        """
        just a plotting function
        """
        fig = plt.figure()
```

```
        plt.rc('text', usetex=True)
        plt.rc('font', family='serif')
        plt.rcParams.update({'font.size': 20})
        fig.set_size_inches(10,5)
        for i in done_jets_list+to_do_jets_list :
                i.plot()
        plt.xlabel(r'$\eta$')
        plt.ylabel(r'$\phi$')
        plt.ylim(-3,3)
        plt.xlim(-2.5,2.5)
        plt.tight_layout()
        plt.savefig("mylittlefigure.png",dpi=600)


##### EXECUTION #####

#defining the algorithm used (kt or antikt)
alpha = -2

#initial p4s
initial_particles_p4 = np.concatenate(databis,axis=0)

#deleting non existant particles (-99.)
to_delete = []
for i,j in enumerate(initial_particles_p4) :
        if np.isclose(j[-1],-99.):
                to_delete.append(i)

initial_particles_p4 = np.delete(initial_particles_p4,to_delete,0)

#Initial distances between particles
distance_matrix=spatial.distance.cdist(initial_particles_p4, initial_particles_p4, distance)

#Define the list of Jets, initialy one for each particle
to_do_jets_list = [Jet(p4init,particle=i) for i,p4init in enumerate(initial_particles_p4)]
done_jets_list = []
done_final_jets_list = []

while len(to_do_jets_list) > 1 :
        a=np.argwhere(distance_matrix == np.min(distance_matrix))[0] #find minimum of distance
        i,j=a
        distance_to_beam = np.vectorize(lambda jet: jet.pt ** alpha)(to_do_jets_list)
        b=int(np.argwhere(distance_to_beam == np.min(distance_to_beam)))
        if distance_matrix[i,j] > distance_to_beam[b] :
                done_final_jets_list.append(to_do_jets_list.pop(b))
                distance_matrix=np.delete(distance_matrix,b,0)
                distance_matrix=np.delete(distance_matrix,b,1)
        else :
```

```
                combine(i,j)

to_do_jets_list += done_final_jets_list
final_jets_particles = [jet.particles_list() for jet in to_do_jets_list]

if __name__=="__main__":
        for i in to_do_jets_list:
                print(i.particles_list())
        plot_jets()
```