Université de Montpellier
Faculté des Sciences

Master Sciences et Numérique pour la Santé
Parcours Bioinformatique, Connaissances, Données

Rapport de stage court HMSN208

Projet de classification de sondes moléculaires pour la détection de l'ARN in situ

Matéo Meynier

Encadré par **Charles Lecellier**

Sous la tutelle de **Thérèse Commes**

Mai 2020 - Juillet 2020

Table des matières

1 Remerciements **2**

2 Introduction **3**
2.1 Contexte du stage . . . 3
2.2 Introduction au projet . . . 3
2.3 Problématique . . . 4
2.4 Objectifs du stage . . . 5

3 État de l'art **5**
3.1 Perceptron . . . 5
3.2 Réseaux de neurones . . . 6
3.3 Apprentissage . . . 7
3.4 Réseaux de neurones à convolution . . . 8
3.5 Modèle sélectionné et données utilisées . . . 9
3.5.1 Pré-traitement de la séquence . . . 9
3.5.2 Modèle utilisé . . . 9
3.5.3 Description des données . . . 10

4 Conception **11**

5 Résultats **16**

6 Second projet **21**

7 Conclusion **22**

8 Bibliographie **22**

1 Remerciements

Je souhaite remercier en premier l'équipe qui m'a accueillie durant mon stage et qui a toujours été derrière moi malgré les conditions exceptionnelles et le télétravail. Ils ont su m'aiguiller et me diriger toujours dans la bonne direction. Je souhaite remercier en particulier Mathys Grapotte et Élodie Simphor qui m'ont accordé de leur temps pour me conseiller et me renseigner sur mon travail. Enfin, je souhaite remercier Charles Lecellier qui m'a permis de travailler sur un sujet qui m'intéressait depuis longtemps, qui a bataillé pour que mon stage ne soit pas annulé et qui m'a permis de découvrir réellement la bio-informatique et la recherche sur ce domaine.

2 Introduction

2.1 Contexte du stage

J'ai effectué mon stage de 1ère année de Master BCD dans l'équipe **IGMM/LIRMM/IMAG Régulations Génomiques Computationnelles** sous la tutelle de Charles Lecellier.

Mon projet est une collaboration entre l'équipe *Régulations Génomiques Computationnelles* et une autre équipe de l'IGMM, l'équipe *Biogenèse des ARNs* sous la direction d'Edouard Bertrand.

Dans le cadre de mon stage, je travaillerai sur des sondes ADN développées dans le cadre du projet smiFISH par l'équipe Biogenèse des ARNs. Le projet sera alors de développer un modèle d'apprentissage automatique sous la direction de l'équipe Régulations Génomiques Computationnelles pour classifier la performance de ces sondes.

2.2 Introduction au projet

Le smFISH ou 'single molecule Fluorescence In Situ Hybridization' est une technique de biologie moléculaire et d'imagerie qui se base sur l'hybridation in situ et la complémentarité des bases nucléiques pour étudier l'expression des gènes dans des cellules individuelles. Le niveau d'ARN est représentatif de l'expression génique et le smFISH permet de détecter et de compter les molécules d'ARN individuelles.

Le principe repose sur un ensemble de sondes composé de plusieurs oligonucléotides d'ADN courts marqués avec des fluorophores. Ce sont ces sondes qui vont s'hybrider avec l'ARN cible car elles sont complémentaires avec celui-ci. Plusieurs sondes complémentaires à la même molécule d'ARN seront requises pour augmenter le signal de détection. La lumière émise provenant du fluorophore d'une seule sonde est faible mais si plusieurs sondes avec le même fluorophore donc la même couleur émettent alors le signal sera bien plus fort et pourra être enregistré.
Le smiFISH (single molecule inexpensive FISH) est une technique dérivée du smFISH développé en partie par l'équipe Biogenèse des ARNs à l'Institut de Génétique Moléculaire de Montpellier.

Dans la technique du smiFISH, nous avons deux types de sondes : des sondes primaires non marquées et des sondes secondaires marquées

Les sondes primaires non étiquetées contiennent à la fois la séquence partagée (FLAP) et une séquence de ciblage spécifique du gène. Les sondes secondaires, elles, sont marquées avec des fluorophores. Les deux types de sondes sont ensuite hybridées ensemble via la séquence FLAP.

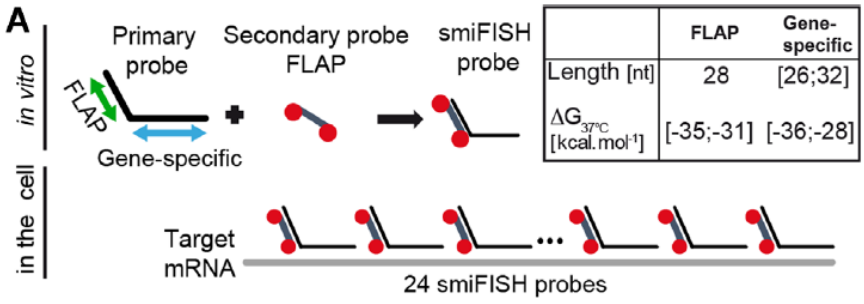

FIGURE 1 – Figure des sondes smiFISH tirée de l'étude [1]

La structure des sondes primaires se présente ainsi :

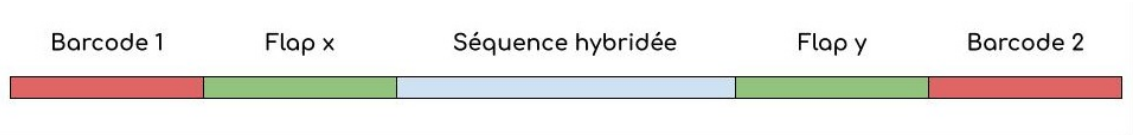

FIGURE 2 – Structure d'une sonde primaire

La séquence qui s'hybride avec l'ARNm est bien présente encadrée par les deux séquences FLAP permettant la fixation avec les sondes secondaires. La sonde est générée puis amplifiée par PCR à partir des barcodes aux extrémités.

Une fois les sondes hybridées avec les gènes d'intérêt arrive l'étape d'analyse des images microscopiques. Certaines sondes non hybridées émettent quand même un signal lumineux et un bruit de fond sera présent sur les images comme sur l'image en dessous.
Pour éviter que les signaux dû à l'hybridation entre des sondes et un ARNm soient masqués par des signaux de sondes non hybridées que nous avons une multitudes de sondes par ARNm. Ainsi le signal lumineux pour la détection d'un ARNm est bien visible et détecter les vrais positifs des faux positifs est plus facile.

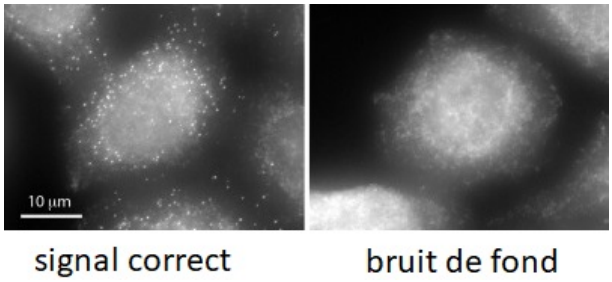

FIGURE 3 – Analyse du signal émis tiré de [1]

2.3 Problématique

Cependant, un problème apparaît. L'efficacité des sondes d'oligonucléotides varie beaucoup en fonction de l'ARNm à détecter. Certaines molécules d'ARNm cible sont détectés facilement alors que d'autres ne renvoient aucun signal. Or la sélection des sondes qui sont des séquences d'ADN ne se basent que sur leur taille, leur température de fusion et leur énergie libre de Gibbs.

Pour améliorer la performance des sondes, il faut donc savoir ce qui influe sur ces dites performances.
L'une des possibilités pour déterminer les facteurs influents est d'utiliser un modèle d'apprentissage automatique et plus précisément, un modèle de classification supervisé. C'est une méthode où un modèle d'apprentissage automatique va être entraîné sur des données d'entrée et de sortie connues pour qu'il puisse ensuite prédire les futurs résultats.

Dans notre cas, quand le modèle sera entraîné, nous lui fournirons une sonde donc une séquence d'ADN et il nous prédira sa performance car il se sera entraîné sur beaucoup de sondes en amont en essayant de prédire leur performance, en analysant la différence entre sa prédiction et la réalité puis en se corrigeant.
Ainsi, dès que le modèle se trompe, il modifie ses paramètres pour prédire le bon résultat d'où le nom d'apprentissage automatique.

La performance ici est la qualité du signal des sondes, le modèle devra prédire le signal que renverront les sondes qu'on lui passe. La qualité du signal renvoyé est classé en 3 catégories : bruit de fond, pas de signal ou signal correct.
La catégorie bruit de fond désigne un signal de mauvaise qualité donc une mauvaise performance de la sonde alors que la catégorie pas de signal désigne le fait qu'aucun signal n'a été détecté.

Élodie Simphor, une M2 BCD effectue son stage de 6 mois sur ces mêmes données mais en ayant une autre mission.
En partant des variables descriptives des sondes fournies par l'équipe Biogenèse des ARNs, elle devait mettre en oeuvre une méthode de classification basée sur de la régression logistique afin de déterminer si les variables fournie sont pertinentes pour discriminer la performance des sondes. Un autre objectif de son stage était de tester une méthode appelée DExTER [2] basée sur des techniques de machine learning afin de sélectionner au niveau des séquences ADN directement, les variables nucléotidiques pertinentes pour discriminer le signal qualité des sondes.

Avant que j'intègre l'équipe, elle a mis en évidence que la catégorie pas de signal est liée au fait que le gène n'était pas exprimé donc il n'y avait aucune hybridation pour ces sondes.
Ainsi nous nous retrouvons plus qu'avec deux catégories vu que la catégorie "pas de signal" n'est pas lié à la performance des sondes : la catégorie "bruit de fond" (0) et la catégorie "signal correct" (1).

Le modèle d'apprentissage automatique s'entraînera sur les données fournies par l'équipe Biogenèse des ARNs pour prédire une fois entraîné, la qualité du signal des sondes qui lui seront données en entrée. Il donnera une prédiction , soit la catégorie 0 soit la catégorie 1.

2.4 Objectifs du stage

Mon objectif pour ce stage est de développer un modèle d'apprentissage automatique pour le problème de classification des sondes de smiFISH.
Plus précisément, l'approche pour le modèle sera un réseau de neurones à convolution (Convolutional Neural Network ou CNN), une approche de deep learning qui a déjà montré son efficacité sur des séquences d'ADN comme DeepSea[3], un CNN prédisant les effets d'une altération des séquences sur la chromatine, Xpresso[4] prédisant l'abondance de l'ARNm uniquement à partir de la séquence du promoteur ou encore Basenji[5] .
Ainsi l'équipe dans laquelle je travaille souhaitait observer comment le réseau de neurones à convolution allait réagir à notre problématique.

Une fois le modèle mis en place et bien entraîné, l'autre objectif sera d'extraire les caractéristiques ou features que le modèle a découvert dans les séquences, les particularités qui lui permettent de prédire la bonne catégorie pour chaque sonde.

La dernière étape sera la comparaison des caractéristiques extraites du CNN et celles de DExTER, la méthode sur laquelle Élodie travaille pour observer si ce sont les mêmes variables détectées ou non.

3 État de l'art

Le réseau de neurones à convolution est un type de réseau de neurones particulier. L'intérêt des réseaux de neurones est qu'on peut lui fournir nos séquences d'ADN directement sans devoir sélectionner des variables d'intérêt. Le réseau lui-même va identifier les caractéristiques importantes pour la qualité des sondes.

3.1 Perceptron

A la base d'un réseau de neurones, il y a un perceptron ou "neurone artificiel". C'est l'unité de base de ce genre de réseau.
Un perceptron prend plusieurs entrées binaires $x_1, x_2, .., x_n$ et produit une seule sortie , y.

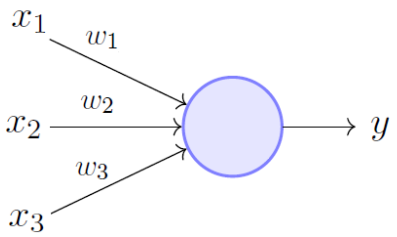

FIGURE 4 – Schéma d'un perceptron tiré du site towardsdatascience

Chacune de ces entrées a un poids w_n, un nombre entier qui représente l'importance de cette entrée, une sorte de coefficient pour résumer.

La sortie brute du perceptron est la somme pondérée $\sum_{i=1}^{n} \boldsymbol{w_i x_i} + \boldsymbol{b}$

Le b dans l'équation correspond à un un paramètre supplémentaire appelé le biais. Il va s'ajouter à la sortie du neurone et va permettre d'ajuster la sortie en même temps que la somme pondérée des entrées du neurone.

Une fonction d'activation est ensuite appliquée sur cette sortie et déterminera si le perceptron est actif (y>0) ou non (y=0).

Les fonctions d'activation les plus utilisés sont la fonction sigmoïde et la fonction reLU. La fonction sigmoïde va renvoyer une valeur comprise entre 0 et 1 alors que la fonction reLU renvoie $max(0, z)$ où z est la somme pondérée présentée au dessus.

3.2 Réseaux de neurones

Ainsi les réseaux de neurones sont composés de couches superposées de neurones reliés les uns aux autres comme montré sur la figure suivante :

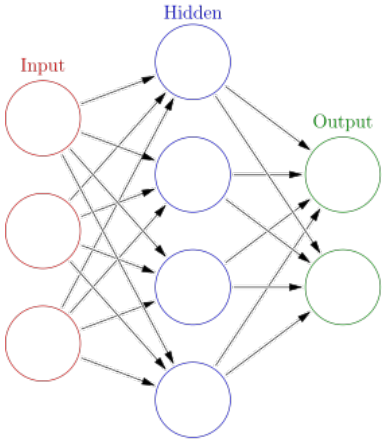

FIGURE 5 – Schéma d'un réseau à neurones tiré de Wikipédia

Ce type d'architecture est le type d'architecture classique pour les réseaux de neurones, c'est un réseau de neurones entièrement connecté. Chaque neurone d'une couche n est connecté à chaque neurone de la couche $n-1$ et de la couche $n+1$. Ainsi il reçoit en entrée chaque sortie des neurones de la couche précédente et retourne sa sortie à chaque neurone de la couche suivante. La première couche est appelée la couche d'entrée, la dernière est la couche de sortie et les couche entre ces deux-là sont les couches cachées.

La couche d'entrée est celle qui va recevoir les données à traiter que ce soit des images ou des profils d'individus.

La couche de sortie, elle, est constituée d'autant de neurones que de classes possible. Si nos données peuvent être classées en 10 classes, on aura 10 neurones qui renverront chacun une valeur qui sera interprétée comme une

probabilité, ainsi le neurone renvoyant la plus grande valeur est la classe prédite par le modèle. Dans un problème de régression, on aura un seul neurone qui renverra une valeur. Les couches cachées représentent la complexité du réseau ou sa finesse. Plus il y a de couches cachées, plus le réseau pourra apprendre et prédire pour des des problèmes complexes car plus de couches veut dire plus de paramètres (poids, entrées) donc plus d'adaptation au problème.

3.3 Apprentissage

Pour que le réseau de neurones s'adapte au problème soumis, il faut l'entraîner ce qui veut dire, donner la bonne valeur aux paramètres du modèle pour qu'il prédise correctement une fois l'entraînement terminé.

Pour l'entraîner, nous allons lui fournir nos données (les sondes ADN), il prédira un signal (0 ou 1) et nous le corrigerons avec la catégorie réelle.

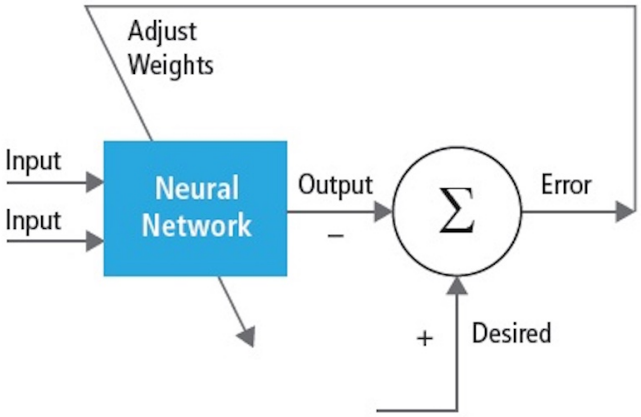

FIGURE 6 – Entraînement d'un réseau de neurone

La première étape de la correction est la fonction de coût : cette fonction va calculer l'erreur entre le résultat prédit et le résultat réel.
Le but étant que la valeur renvoyée par cette fonction soit la plus petite possible et c'est en faisant évoluer les paramètres du modèle que ce soit les poids ou les biais qu'on y arrive.

L'algorithme permettant de déterminer la meilleure valeur pour les poids et les biais du réseau est l'algorithme de rétro-propagation, qui utilise le principe de la descente de gradient. Le gradient va nous indiquer la direction de plus grande variation pour une fonction qui correspond à sa pente la plus forte. Ici la fonction que prend le gradient est notre fonction de coût. Pour trouver le minimum, il faudra alors se déplacer dans la direction opposée au gradient.
L'algorithme va se déplacer dans cette direction opposée en modifiant les poids et les biais couche par couche, de la dernière couche jusqu'à la première.

Ainsi pour chaque exemple donné, nous avons une "forward pass", une passe vers avant qui aboutit à une prédiction ensuite le calcul de l'erreur entre cette prédiction et la valeur réelle puis une "backward pass", une passe à l'envers où les poids et les biais des couches sont modifiés pour trouver le minimum de la fonction de coût.

Ainsi le réseau de neurones s'entraîne bien lorsque le résultat de la fonction de coût baisse bien au fil des exemples données.
Mais il faut se méfier aussi que celle-ci ne descende pas trop bas car ça pourrait être synonyme de sur-entraînement du réseau. Cela arrive quand le modèle s'est trop entraîné sur les données et qu'il est devenu trop spécifique à ces données-ci en ne généralisant pas pour le problème en lui-même. Deux solutions seront expliquées plus tard dans le rapport pour éviter ce problème, le dropout au niveau de l'architecture et l'early-stopping au niveau de

l'entraînement.

3.4 Réseaux de neurones à convolution

Le réseau de neurones à convolution a un type d'architecture différent comparé au réseau de neurones entièrement connecté car il est en fait constitué de deux blocs :
le premier bloc va prendre ce qui a été donné en entrée du réseau donc une image et va extraire les caractéristiques de celle-ci en lui apposant des filtres et en la réduisant à l'essentiel en plaçant les informations les plus importantes dans une matrice.
Cette matrice sera l'entrée du deuxième bloc qui est en fait un réseau de neurones entièrement connecté comme vu précédemment qui prédira le résultat.

Les particularités du premier bloc sont les opérations de convolution et de pooling.
Un filtre est déplacé sur l'image et on applique un produit scalaire entre la zone de l'image traitée et le filtre lui-même. On obtient alors une image de taille légèrement réduite.

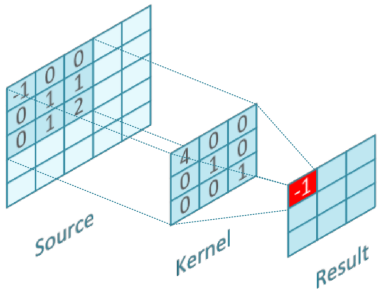

FIGURE 7 – Convolution

Ainsi l'opération de convolution consiste à appliquer un filtre sur l'image entrée. Ce filtre va permettre de faire ressortir des caractéristiques dans l'image comme une forme, un contour ou une couleur. Si un filtre est crée pour détecter les formes, lorsqu'il va être appliqué sur une zone de l'image qui possède une forme, le produit scalaire entre cette zone de l'image et le filtre va être élevé et sera le signe de la détection d'une forme Plusieurs filtres peuvent être appliqués pour détecter plusieurs caractéristiques.

L'opération de pooling consiste à réduire l'image obtenue après la convolution en conservant les valeurs les plus grandes des pixels.

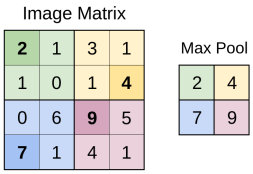

FIGURE 8 – Méthode du max pooling

Ainsi on retient les caractéristiques importantes de l'image tout en simplifiant celle-ci.

La convolution met en lumière les caractéristiques recherchées dans une image via le filtre et le pooling simplifie l'image en sauvegardant les caractéristiques détectées. Il faut noter aussi que plusieurs cycles de convolution/pooling peuvent s'enchaîner pour plus de finesse.

Ensuite cette image simplifiée est données en entrée au réseau de neurones entièrement connecté pour la prédiction finale.

Les réseaux de neurones à convolution grâce à ce premier bloc sont plus pertinent qu'un réseau de neurones entièrement connecté "simple" pour analyser des images et prédire par exemple si des chats ou des chiens apparaissent dessus.

La majorité des informations des précédentes sections proviennent de mon rapport de projet bibliographique durant le semestre 1. Je m'étais renseigné sur les réseaux de neurones grâce au livre en ligne *"Neural Networks and Deep Learning"* [6] de Michael Nielsen, chercheur en informatique ainsi que de l'article *"Deep learning : new computational modelling techniques for genomics"*[7] discutant des applications des réseaux de neurones à convolution pour la génomique .

3.5 Modèle sélectionné et données utilisées

3.5.1 Pré-traitement de la séquence

Une étape importante avant de pouvoir utiliser un réseau de neurones à convolution pour nos séquences d'ADN est de transformer ces dernières en images sinon impossible de les traiter.

Pour cela, nous utilisons l'encodage one-hot. Chaque position de la séquence est modélisé sous la forme d'un vecteur de 4 éléments dont un seul est mis à 1. Il indique quelle base se trouve à cette position.

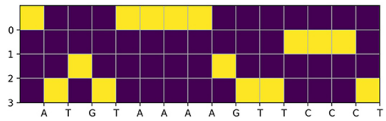

FIGURE 9 – Exemple d'un vecteur encodé one-hot pour un fragment d'ADN de 15bp

Si c'est A qui est présent alors le vecteur est de cette forme : [1,0,0,0].

La séquence est interprétée par le réseau comme une image de taille n*4 où n représente la longueur de la séquence considérée.

3.5.2 Modèle utilisé

Pour gagner du temps, je me suis inspiré des travaux de Mathys GRAPOTTE, doctorant travaillant dans notre équipe qui pour son stage de M2 BCD *"Extraction des paramètres pour comprendre les prédictions effectuées par apprentissage profond"* avait dû réalisé un réseau de neurones à convolution. La figure ci-dessous présente l'architecture du réseau de neurone défini par Mathys GRAPOTTE et l'équipe Régulations Génomiques Computationnelles.

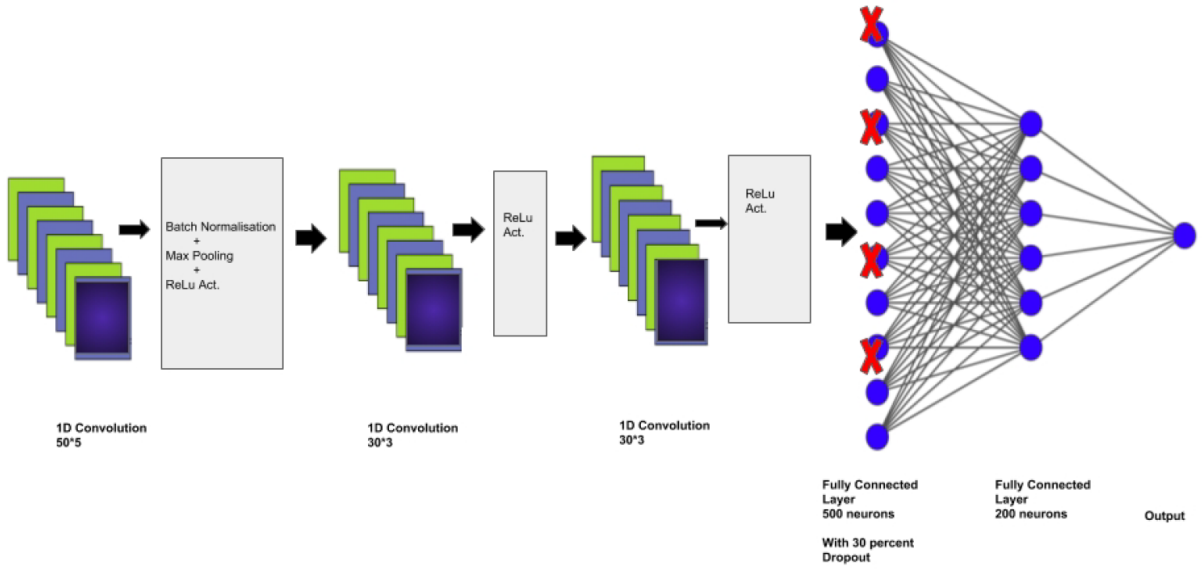

FIGURE 10 – Schéma du réseau de neurones à convolution développé par l'équipe [8]

Il est constitué de trois couches de convolution et de deux couches entièrement connecté pour aboutir à une couche de sortie avec un seul neurone car c'était un problème de régression.

J'ai donc utilisé la même architecture pour mon réseau de neurones à convolution à part que pour la couche de sortie du réseau, deux neurones sont présent pour représenter les deux catégories de signal possible, signal 0 ou signal 1.

3.5.3 Description des données

Les fichiers de données fournis étaient au nombre de deux.
Le fichier *"geneName_sig_all.csv"* contient le nom du gène numéroté et le signal correspondant.
A savoir qu'il y a plusieurs sondes par gène mais un gène n'a qu'un signal correspondant vu que soit le gène est bien détecté grâce aux multiples sondes et son signal est de 1 sinon il est de 0.

gene	sig
NFKBIA_0	1
NFKBIA_1	1
NFKBIA_2	1
NFKBIA_3	1
NFKBIA_4	1
NFKBIA_5	1
NFKBIA_6	1
NFKBIA_7	1
NFKBIA_8	1
NFKBIA_9	1

FIGURE 11 – fichier geneName_sig_all.csv

Le fichier *"geneName_seq_all.fa"* contient le nom du gène numéroté et la séquence ADN entière (barcodes + FLAP + séquence hybridée).

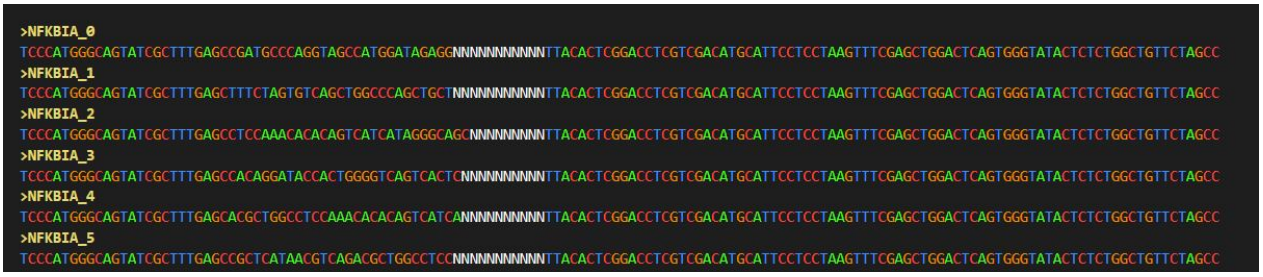

FIGURE 12 – fichier geneName_seq_all.fa

4 Conception

Pour concevoir le réseau de neurones à convolution (abrégé CNN dans la suite du rapport), j'ai été aidé d'Elodie Simphor pour la récupération et la gestion des sondes ADN et de Mathys Grapotte pour réaliser mon CNN.
J'ai notamment suivi ce qu'avais fait Mathys pour son propre CNN mais j'ai préféré ne pas recopier son code bêtement au risque de ne rien comprendre après.

Le code est disponible sur mon GitHub à l'adresse : `https://github.com/mateo-meynier/stageM1`
Pour ce projet de classification de sondes ADN, il y a deux fichiers python : *preloadCNN.py* et *CNNmodel.py*
Le fichier **preloadCNN.py** contient le pré-traitement des données qui est expliqué un peu plus loin dans le rapport.
Le fichier **CNNmodel.py** contient le CNN, le chargement des données, l'entraînement et le test du CNN.

La classe Kbar provient du GitHub `https://github.com/yueyericardo/pkbar` et permet d'afficher une barre de progression durant l'entraînement du modèle.

La classe EarlyStopping provient du GitHub `https://github.com/Bjarten/early-stopping-pytorch` et implémente l'early stopping évoqué dans la section 3.3 Apprentissage.

Avant tout, pour coder le CNN, j'ai eu le choix d'utiliser l'une des deux bibliothèque logicielle Python d'apprentissage machine, PyTorch et Keras.
Mathys m'a conseillé PyTorch vu que c'était vers cette bibliothèque qu'il s'était tourné l'année dernière et que d'après lui, elle est moins "boite noire" que Keras mais qu'il faudrait alors du temps pour bien la prendre en main.

FIGURE 13 – Logo de PyTorch, bibliothèque logicielle Python open source

PyTorch propose deux atouts majeurs :

- la manipulation de tenseurs (tableaux multidimensionnels), la conversion numpy/tenseurs et la possibilité d'effecteur les calculs soit sur CPU soit sur GPU
- le calcul des gradients pour faciliter le travail de l'algo de rétro propagation

Pré-traitement des données

[Code disponible dans le fichier preloadCNN.py]

Pour mettre sur pied un CNN, la première étape est de pré-traiter les données.

Comme je l'ai détaillé précédemment, je possédais deux fichiers fournis par Élodie, le fichier geneName_sig_all.csv avec le nom du gène et le signal correspondant et le fichier geneName_seq_all.fa avec le nom du gène et la séquence de la sonde.
La séquence est bien la séquence entière Barcodes + Flap + Séquence hybridée mais vu que les séquences hybridées diffèrent de taille, elles sont donc complétées par des 'N' pour que la taille des séquences soit la même et ne pose pas de problème au CNN.

Pour obtenir un fichier utilisable, il a fallu faire une jointure entre ces deux fichiers grâce à la colonne commune du gène. Ensuite j'ai découpé les données en un set d'entrainement (trainset) et un set de test (testset). La répartition est 80% pour le set d'entrainement et 20% pour le set de test.

Le script réalisant le pré traitement est le fichier preloadCNN.py A la fin, on obtient trois fichiers numpy pour chacun des set, un fichier genenames.npy, un fichier sequences.npy et un fichier signals.npy qui contiennent respectivement les noms des gènes, les séquences et la qualité du signal pour le set. Les séquences auront été transformées en "images" via le one hot encoder comme expliqué dans la section 3.5.

Définition du CNN

[Classe CnnModel disponible dans le fichier CNNmodel.py]

Une fois les données mises en forme pour pouvoir etre traitré par le CNN, il faut définir son architecture.

J'ai adapté l'architecture mise en place par Mathys pour traiter un problème de classification plutôt qu'un problème de régression.
Il a fallu changer la couche de sortie et y mettre deux neurones à la place d'un seul ainsi les deux neurones renverront une valeur interprétée comme une probabilité et la plus grosse valeur définira alors la classe à laquelle appartient la sonde donnée au modèle.

```
def __init__(self):
    super(CnnModel, self).__init__()
    self.conv1 = nn.Conv2d(1, 50, (5, 4), bias=False)
    self.batch1 = nn.BatchNorm2d(50, track_running_stats=False)
    self.maxpool1 = nn.MaxPool1d(2)
    self.conv2 = nn.Conv1d(50, 30, 3, bias=False)
    self.conv3 = nn.Conv1d(30, 30, 3, bias=False)
    self.flatt = nn.Flatten()
    self.dense1 = nn.Linear(1980, 500)
    self.dropout1 = nn.Dropout(p=0.3)
    self.dense2 = nn.Linear(500, 200)
    self.dense_output = nn.Linear(200, 2)
```

FIGURE 14 – Définition de l'architecture du CNN

Les trois opérations de convolution y sont présentes (1 Conv2D et 2 Conv1D), BatchNorm2d va normaliser les données entrées et MaxPool1D est l'opération de pooling comme expliqué section 3.4.
L'opération Flatten permet de passer du bloc de convolution/pooling au bloc du réseau entièrement connecté en transformant les données sous forme d'un tableau 1 dimension.
Les données passent ensuite dans une fonction linéaire puis subissent une opération de dropout qui consiste à remplacer certaines données par 0 de façon aléatoire avec une probabilité de 0.3. Cette opération est l'une des solutions pour diminuer le risque de sur-entraînement.
Les deux dernières opérations sont des transformations linéaires des données pour arriver aux deux neurones de sorties.

Chargement des données

[Classe SeqDataset et fonction load_data disponible dans le fichier CNNmodel.py]

Avant de détailler le processus de chargement des données, je tiens à préciser comment vont être découpées celles-ci.
A cette étape là, on a déjà un set d'entraînement et un set de test mais on va de nouveau découper le set d'entraînement en un set de validation et un set d'entraînement.

Le concept derrière est la validation croisée : lors de l'entraînement du réseau, une fois qu'on a fourni au réseau toutes les données du set d'entraînement, qu'il a fait sa prédiction et que l'algorithme de rétro-propagation a fait son travail, on lui donne en entrée ce set de validation. De nouveau, il prédit le résultat pour chacune des données de ce set et la fonction de coût et l'algorithme de rétro-propagation le corrige.
On répète ce cycle autant de fois qu'on passe sur le set d'entraînement.

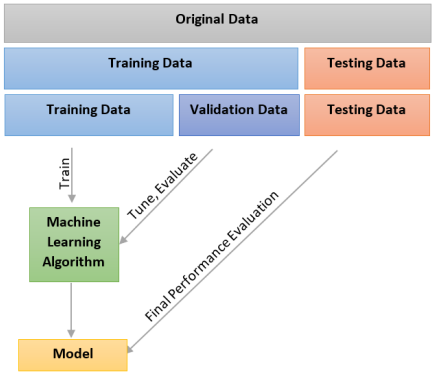

FIGURE 15 – Concept de la validation croissée

La validation croisée peut être couplée à une autre méthode, l'early stopping :
On contrôle le résultat de la fonction de coût pour le set de validation et on sauvegarde le modèle tant que ce résultat continue à diminuer. Dès qu'il commence à augmenter, au bout d'un certain seuil défini par l'utilisateur, on stoppe l'entraînement.
C'est la deuxième mesure pour contrer le sur-entraînement.

Revenons au chargement des données.
Pour les charger, on utilise la classe Dataset de PyTorch qui permet de définir un jeu de données. Un échantillon du Dataset sera un dictionnaire Python ['seq' :seq, 'signal' :signal, 'genename' :genename].
Ma classe Dataset personnalisée hérite de Dataset et surcharge ses méthodes : c'est la classe SeqDataset.

La fonction load_data est la fonction qui va charger les données via SeqDataset :
En premier, elle va charger les sets d'entraînement et de test enregistrés sous forme de fichier numpy. Ensuite, elle définit des instances de SeqDataset pour créer le dataset d'entraînement et le dataset de test.

```
train_data = SeqDataset(seq_path=seqs_trainset_path, genenames_path = genenames_trainset_path,
                        signals_path = signals_trainset_path )
test_data = SeqDataset(seq_path = seqs_testset_path, genenames_path=genenames_testset_path,
                        signals_path = signals_testset_path)
```

FIGURE 16 – Définition des datasets

Pour itérer sur les datasets, on utilise l'objet DataLoader fourni par PyTorch.

```
train_sampler = SubsetRandomSampler(train_idx)
val_sampler = SubsetRandomSampler(val_idx)

train_loader = DataLoader(dataset=train_data, shuffle=False, batch_size = batch, sampler=train_sampler)
valid_loader = DataLoader(dataset = train_data, shuffle=False,batch_size=batch, sampler= val_sampler)
test_loader = DataLoader(dataset=test_data, shuffle=True, batch_size = batch)
```

FIGURE 17 – Définition des dataloader

Le DataLoader va regrouper les données sous forme de batch de taille n précisé par l'utilisateur et les mélanger via l'échantillonneur SubsetRandomSampler.

Quand on itérera sur le DataLoader lors de l'entraînement, il renverra un batch de taille n composé de dictionnaire Python ['seq' :seq, 'signal' :signal, 'genename' :genename].

Entraînement du modèle

[Fonction training_pytorch_model disponible dans le fichier CNNmodel.py]

Pour entraîner le CNN, on va itérer sur le dataloader du set d'entraînement.
Ainsi à chaque nouvelle itération, on récupère un batch n de séquences et un batch n de signaux correspondants aux séquences. Ces n séquences et signaux sont stockés dans des Tensors, des matrices multidimensionnelles spécifique à PyTorch, contenant des éléments d'un seul type de données.
On récupère pas les noms des gènes car ils n'ont aucune utilité dans l'entraînement du modèle.
Un epoch définit un passage sur toutes les données donc si nos données sont divisées en 50 batchs de taille 10, un epoch sera fini lorsqu'on sera passé sur les 50 batchs.

La taille du batch et le nombre d'epochs sont des hyperparamètres du modèle, des paramètres dont la valeur est utilisée pour contrôler le processus d'apprentissage et sont définis avant le début de l'apprentissage. En revanche, les valeurs des autres paramètres comme les poids et biais sont gérés par le modèle lui même et vont évoluer durant l'apprentissage du modèle.

On donne les n séquences au modèle qu'on a définit plus tôt et il nous renvoie ses prédictions de signal pour chacune des séquences.
Plus particulièrement, on obtient un Tensor de dimension [batch_size,2] car pour chacune des séquences, nous obtenons deux valeurs, une pour chacun des neurones de sorties.
La valeur la plus élevée indique la classe que le réseau a déterminé comme étant la classe la plus probable. Si la valeur la plus élevée se trouve dans la première case, le réseau prédit la classe 0 (bruit de fond) et inversement si elle se trouve dans la deuxième case, il prédit la classe 1 (signal correct).

Une fois les signaux prédits par le modèle, on fournit les signaux réels et les signaux prédits à notre fonction de coût qui pour le projet est la fonction CrossEntropyLoss.
Celle-ci renvoie ensuite une valeur nous permettant d'estimer l'erreur globale des prédictions et ceci va nous indiquer comment faire évoluer les paramètres du modèle pour diminuer cette erreur.

Pour faire évoluer les paramètres donc les poids et biais du CNN, on doit calculer les gradients de ces dits paramètres en partant de la dernière couche. L'appel à loss.backward le fait automatiquement.
Une fois les gradients calculés, l'évolution manuelle des paramètres se fait ainsi : paramètre -= learning_rate * paramètre.gradient

Le learning rate ou taux d'apprentissage est un autre hyperparamètre qui contrôle dans quelle mesure les poids sont ajustés par rapport au gradient. Plus celui ci est faible, plus la descente de gradient se fait lentement. Cet hyperparamètre détermine en fait le temps qu'on va mettre à converger vers le minimum de notre fonction de coût.

Mais on peut appeler des algorithmes d'optimisation (ex : AdaGrad, RMSProp, Adam) qui donneront de meilleures performances que la version manuelle. Ils sont disponibles via le paquet optim de PyTorch.

Ici j'ai utilisé l'optimiseur ADAM en le définissant dans le main et en l'appelant ensuite via optimizer.step(). Le learning rate est par défaut à 0.001 lors de l'appel à ADAM et a été laissé comme tel.

```
custom_model.train()

for i_batch, item in enumerate(train_loader):
    seq = item['seq'].to(device)
    signal = item['signal'].to(device)

    # Forward pass
    outputs = custom_model(seq)

    signal = signal.long()
    loss = criterion(outputs, signal)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

FIGURE 18 – Entraînement du modèle

Une fois la fin de l'itération sur le set d'entraînement, on passe à l'itération sur le set de validation avec une différence par rapport au set précédent.
À ce niveau là, pas de phase d'optimisation des paramètres, on contrôle juste les variations de la fonction de coût sur ce set de données. On arrêtera l'entraînement si on observe une tendance du résultat de la fonction de coût à remonter via l'early stopping.

```
custom_model.eval()  # Sets the model in training mode, which changes the behaviour of dropout layers...
with torch.no_grad():
    for i_val, item_val in enumerate(valid_loader):
        seq_valid = item_val['seq'].to(device)
        signal_valid = item_val['signal'].to(device)

        outputs_valid = custom_model(seq_valid)
        signal_valid = signal_valid.long()
        loss_v = criterion(outputs_valid, signal_valid)
```

FIGURE 19 – Validation du modèle

Test du modèle et interprétation du résultat

[Fonction test_pytorch_model disponible dans le fichier CNNmodel.py]

La phase de test consiste à déterminer si le modèle est bien entraîné ou non.
Le set de test contient des données que le CNN n'a jamais vu et simule donc des futures données entrées. On ne va pas calculer le résultat de la fonction de coût mais on va compter le nombre de prédictions justes et fausses et déterminer ainsi la précision du modèle. On veut aussi savoir si la précision est la même sur les deux classes ou si une classe se détache et est souvent mieux prédite.

Pour cela, une fonction de sklearn, une bibliothèque Python, existe : classification_report()
En lui fournissant les signaux prédits et les signaux réels, la fonction va mesurer la qualité des prédictions et va renvoyer trois mesures par classe : precision, recall et le f1-score.

Pour chaque classe, la mesure *'precision'* correspond au pourcentage de prédictions positives qui étaient des cas réellement positifs, *'recall'* le pourcentage de cas réellement positifs qui ont été prédits comme tels et *'f1-score'* est une moyenne pondérée des deux mesures précédentes tel que le meilleur score est de 1,0 et le pire de 0,0. Grâce à ces mesures, on peut estimer si la performance du CNN est bonne ou non.

```
Accuracy (classification_report) :  0.9615284772337885

 Signal0 f1-score (classification_report) :  0.8718214841722886
Signal1 f1-score (classification_report) :  0.9773678262728688

 macro AVG precision (classification_report) :  0.9252615528155926
macro AVG recall (classification_report) :  0.9239310244711039
macro AVG f1_score (classification_report) :  0.9245946552225787
```

FIGURE 20 – Résultat de classification_report()

Mais une autre mesure est utile pour estimer la performance d'un classificateur binaire, c'est la courbe ROC (pour receiver operating characteristic), aussi appelée courbe sensibilité/spécificité.
C'est une courbe qui donne le taux de vrais positifs (fraction des positifs qui sont effectivement prédits comme tel) en fonction du taux de faux positifs (fraction des négatifs qui sont incorrectement prédits comme positifs).

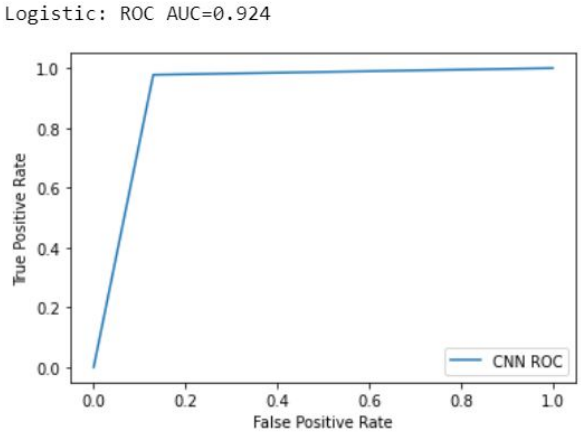

FIGURE 21 – Courbe ROC

Si la courbe coïncide avec la diagonale, le modèle est comparable à un modèle aléatoire donc un mauvais modèle. Mais plus la courbe est proche du coin supérieur gauche du graphe, meilleur est le modèle car le modèle a un très haut taux de vrais positifs et un faible taux de faux positifs.
Pour comparer des modèles entre eux, on peut utiliser l'AUC (Area Under Curve) qui est donc l'aire sous leur courbe ROC respective. Plus l'AUC est proche de 1, plus la courbe ROC est à proximité du coin gauche du graphe, plus le modèle est performant pour classifier.

5 Résultats

Après la conception du CNN, les étapes suivantes sont l'entraînement et le test.

J'ai ainsi testé le modèle avec les données fournies par l'équipe de biologistes.
Je possédais 96 305 données sous la forme [nom du gène, séquence, signal] dont 81 923 dont le signal appartenait à la classe 1 (signal correct) et 14 383 appartenant à la classe 0 (bruit de fond).Le set de train contenait 76 062 données (train/ valid) et le set de test, 20 243.

On peut noter qu'il y déséquilibre entre les classes (85% de signaux 1 et 15% de signaux 0). Ainsi le CNN va s'entraîner presque 6 fois plus sur des séquences liées à un signal 1 ce qui peut poser problème.

Pour comprendre les résultats du modèle, il faut déjà observer ce qu'a obtenu Élodie sur ces mêmes sondes. Elle travaillait sur ces mêmes données de son coté via la méthode DEXTER (Domain Exploration To Explain gene Regulation).
Cette méthode va identifier des paires de k-mers, des régions pour lesquelles il existe une correspondance entre la classe du signal qualité et la fréquence du k-mer dans la région définie de chaque sonde. Les paires identifiées comme importantes sont combinées pour permettre de prédire la classe 0 ou 1 du signal.

Une fois ces variables identifiées, DEXTER va apprendre un modèle qui prédit le signal qualité sur la base de ces variables mais aussi renvoie deux matrices contenant des dinucléotides associés à des pourcentages qui permettent d'extraire les variables identifiées.

Premièrement, le travail d'Élodie a mis en lumière qu'il fallait mieux travailler sur les séquences ADN entières donc barcodes + flap + séquence hybridée car lorsqu'elle fournissait à DEXTER seulement la séquence hybridée de la sonde (environ 24 bp) , l'AUC du modèle apprit par DEXTER ne dépassait pas 0,55 ce qui correspond à un modèle aléatoires.
Donc ces modèles entraînés seulement sur les séquences hybridées ne sont pas assez discriminants pour bien classifier les sondes. Alors que si DEXTER travaille sur les séquences entières, l'AUC du modèle peut monter jusqu'à 0.85 ce qui correspond à un bon modèle de classification.

C'est donc pour cela que j'ai fourni à mon CNN les séquences entières car les résultats précédents indiquent que l'information permettant de discriminer une sonde ne se trouve pas totalement dans la séquence hybridée seule.

La deuxième information importante découverte par Élodie concerne les variables extraites par DEXTER sur la séquence entière.
Elle a répertoriée les 10 premières variables extraites par DExTER dans un tableau en indiquant leurs positions dans les séquences.

Le tableau en dessous est le tableau des variables extraites pour les sondes de taille minimale (130 nucléotides) :

Variables (séquences)	AA	AG	CTAC	GCT	GGC	GTAT	TA	TAT	TGAG	TGAG
Position dans la séquence	1-10	0	10	0-20	10	12-100	0-12	91-100	71-110	91-110
Valeur du coefficient	-1,28	-0.56	-4.45	-0.90	-1.56	-34.12	-0.64	-1.64	-8.86	-11.23

FIGURE 22 – Variables extraites par DEXTER

Si on identifie ces positions sur les sondes de taille minimale, on observe que la majorité des variables extraites par DEXTER se situent au niveau des barcodes entre 0-25bp et 105-130bp ou au niveau des FLAP entre 25-53bp et 77-105bp.

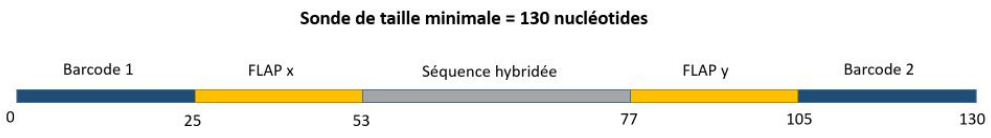

FIGURE 23 – Sonde ADN

Par rapport à cette dernière information, il faut donc que le CNN détecte de lui même que ces variables extraites par DEXTER dans les barcodes sont les variables permettant de discriminer les sondes.

L'objectif alors est que le CNN identifie dans les barcodes, des motifs particuliers qui engendrent un bon ou mauvais signal durant sa phase d'entraînement et que lors de la phase de test, il analyse des barcodes inconnus et qu'il retrouve ces mêmes motifs lui permettant alors de prédire le bon signal.

Un barcode est spécifique à un gène ce qui veut dire que toutes les sondes censé détecter GATA1 par exemple, posséderont les mêmes barcodes. J'ai alors découpé mon set d'entraînement et mon set de test en réservant 20% des gènes au set de test et donc 20% de tous nos barcodes sont réservés pour le test.

Voici le résultat de la fonction de coût du CNN avec 300 epochs et une taille de batch de 200 :

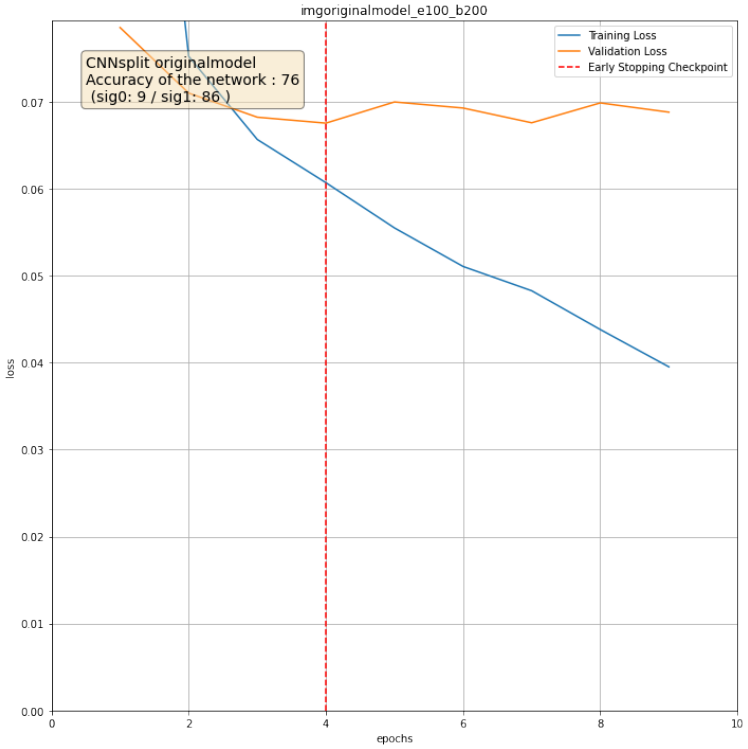

FIGURE 24 – Courbe de la fonction de coût

Comme on le voit sur le graphe, la courbe de la fonction de coût pour l'entraînement (trait bleu) descends bien ce qui montre que le modèle apprend quelque chose. Par contre, la courbe de la validation varie sans vraiment diminuer et le modèle est stoppé dans l'entraînement à l'epoch 10 car celle-ci remontait trop.
La ligne rouge désigne le meilleur epoch, celui avant que la courbe de la validation ne remonte, c'est le modèle à l'epoch qui est sauvegardé puis utilisé pour la phase de test.

Pour la phase de test, les résultats sont décevants :

```
Accuracy (classification_report) :  0.7567436679742361

 Signal0 f1-score (classification_report) :  0.09250225835591688
Signal1 f1-score (classification_report) :  0.859547577104829
```

FIGURE 25 – Précision du modèle

Le CNN n'arrive pas du tout à bien classer les signaux de classe 0 avec une précision de 9%. On se retrouve alors avec un modèle avec une précision globale de 75% car celui-ci classe la majorité des signaux en classe 1.

La courbe ROC met bien en évidence l'inefficacité du modèle avec une AUC de 0.503.

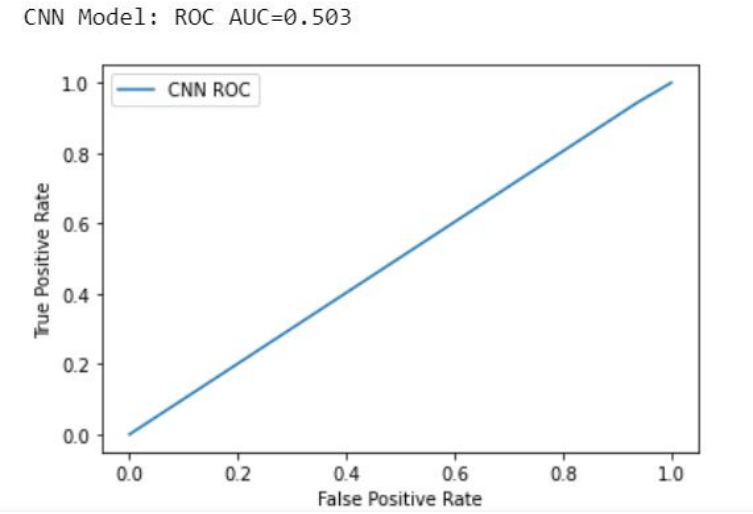

FIGURE 26 – Courbe ROC

Même en faisant varier les epochs et la taille des batchs, les prédictions pour les signaux de classe 0 restent très bas.

Dans un premier temps, j'ai voulu corriger ce déséquilibre de classes que j'ai relevé au début de cette section. Pour résoudre ce problème, il y a trois solutions :

- enlever une part des données avec des signaux à 1 pour rééquilibrer manuellement les classes
- appliquer des poids aux classes via la fonction de coût
  Simplement, pour les classes avec un petit nombre d'exemples d'entraînement, le poids donné sera important afin que le réseau soit davantage sanctionné s'il fait des erreurs de prédiction sur cette classe. Pour les classes avec un grand nombre d'exemples, le poids sera moins important car le réseau a de quoi s'entraîner dessus
- utiliser l'échantillonneur WeightedRandomSampler qui permet de passer des probabilités pour chacune des classes.
  Ces probabilités permettront de favoriser une classe en la faisant apparaître plus de fois dans chaque batch.

Les deux dernières solutions n'ont donné que de piètres résultats sans que je sache réellement pourquoi.
Par manque de temps, je n'ai pas pu déterminer si le problème venait de moi ou si ces deux méthodes n'étaient pas encore au point ce qui est possible vu les nombreuses questions à ce sujet sur les forums.

J'ai donc mis en place la première solution même si elle oblige à enlever beaucoup de données donc une perte importante d'informations.

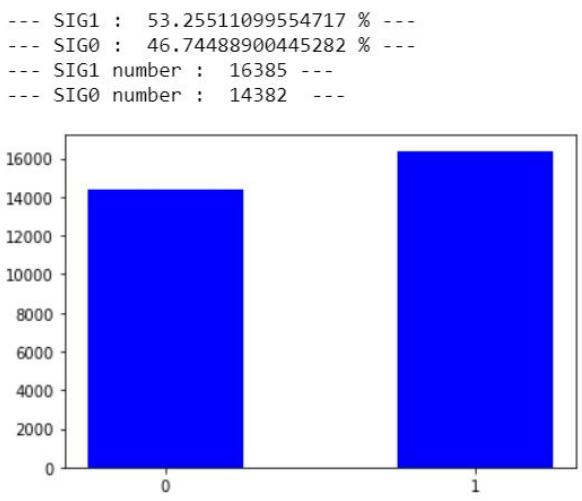

FIGURE 27 – Équilibrage des classes

Ce rééquilibrage des classes a permis d'augmenter la précision des prédictions pour le signal 0 mais la précision des prédictions pour le signal 1 a alors chuté.

```
 --- Résultats du set de test ---

Accuracy (classification_report) :  0.5015178143473399

 Signal0 f1-score (classification_report) :  0.38070662961492 66
Signal1 f1-score (classification_report) :  0.5828877005347594
```

FIGURE 28 – Précision du modèle rééquilibré

Cela a mis en lumière la réelle raison de l'échec du CNN : le manque de données.
Dans les 90 000 séquences que j'ai, il y a seulement 87 paires de barcodes différentes, le barcode 1 et le barcode 2 sont liés entre eux et sont spécifiques à un seul gène.

Or on sait que les variables discriminantes extraites par DEXTER se situent pour la plupart dans les barcodes. Avec seulement 174 exemples de paire de barcodes, le réseau n'a pas assez d'exemples pour analyser les barcodes et identifier ces variables.
De manière générale, les réseau de neurones nécessitent un grand ensemble de données pour pouvoir bien s'entraîner et assimiler les caractéristiques des données.

Pour ce projet de classification de sondes ADN, un modèle aussi complexe que le réseau de neurones à convolution est finalement moins performant que la régression logistique sur laquelle est basé DEXTER.

Ainsi mon projet de développer un réseau de neurones à convolution pour résoudre le problème de classification des sondes de smiFISH a abouti à l'abandon de ce type d'approche pour cette problématique mais cela aura montré que des modèles complexes comme les réseau de neurones ne sont pas des réponses à tout problème, qu'un de leurs désavantages est bien la demande importante de données et que des modèles plus simples comme la régression logistique ont toujours leurs places.

Je tiens à préciser que même si le CNN n'a rien donné pour ma problématique, il a pu donner de très bons résultats dans l'équipe où je travaille notamment le travail de Mathys Grapotte qui a entraîné des CNNs pour prédire des signaux CAGE (Cap Analysis of Gene Expression)[8].

6 Second projet

Vu qu'il restait trois semaines avant la fin de mon stage, mon tuteur Charles m'a donné un autre projet de développement d'un réseau de neurones à convolution en lien avec la technologie CAGE[9].

Le consortium FANTOM5 propose une cartographie de sites de transcription (TSSs) chez plusieurs espèces. Ces TSS ont été obtenus à l'aide de cette technologie CAGE pour Cap Analysis of Gene Expression. À partir de la cartographie des TSS a été développée une mesure de l'expression des gènes.
Ainsi mes données étaient le nom du pic CAGE, le cluster auquel il appartient, le signal CAGE additionné autour du pic et la séquence correspondante.

Mes objectifs étaient de mettre en place un modèle de prédiction global permettant donc de prédire cette mesure de l'expression mais aussi développer des modèles spécifiques à chaque cluster.
Le but de développer un modèle global et des modèles spécifiques était de pouvoir ensuite tester chacun des modèles sur les données des autres modèles et d'observer si un des modèles pourraient se prêter à de l'apprentissage par transfert.

Par exemple, une fois le modèle général bien entraîné, il sera testé sur les données de chaque cluster pour voir si il arrive quand même à bien prédire l'expression.
Il faudra aussi tester chaque modèle spécifique à un cluster sur les données des autres clusters

Le fichier fourni était un fichier fasta d'environ 427 500 séquences avec 20 clusters différents.

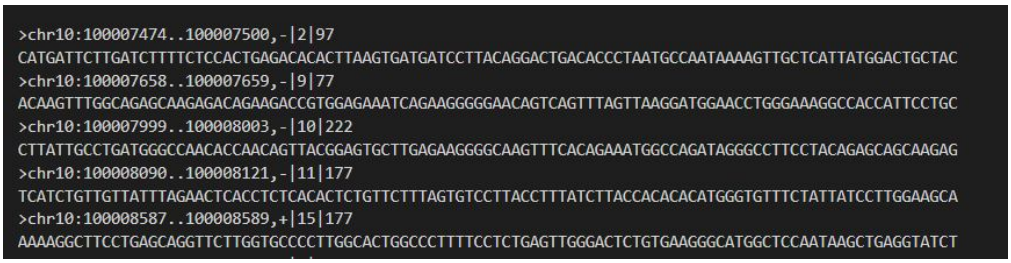

FIGURE 29 – Fichier cluster_101bp_sumQ20

Pour le réseau de neurones à convolution, je devais partir du réseau de neurones à convolution développé par Mathys vu que ce modèle était conçu pour prédire lui aussi un signal CAGE.
J'ai ensuite appliqué les mêmes étapes que lors du premier projet : pré-traitement et chargement des données, entraînement du modèle puis test du modèle.
Le pré-traitement des données a pris plus de temps que dans le précédent projet car il a fallu découper les données en fonction de leurs clusters. Ainsi j'obtenais à la fin de ce traitement pour chacun des clusters, 3 fichiers numpy contenant les séquences, les expressions et les noms des pics CAGE correspondants.

Ensuite il a fallu pour chaque modèle :

- entraîner le modèle sur son set d'entraînement
- tester le modèle sur les 20 sets de test des clusters et du modèle général
- mesurer la corrélation de Spearman et la corrélation de Pearson qui permettait d'évaluer si le modèle était performants sur les sets de test

Un coefficient de corrélation va calculer dans quelle mesure deux variables tendent à changer ensemble et permet de décrire l'importance et le sens de la relation.

La corrélation de Pearson va mesurer si une relation linéaire existe entre deux variables alors que la corrélation de Spearman mesure pour une relation monotone. Une relation est linéaire lorsqu'une modification de l'une des variables est associée à une modification proportionnelle de l'autre variable. Alors que pour une relation monotone, les variables ont tendance à changer ensemble, mais pas forcément à une vitesse constante.

Pour voir le code pour l'entraînement et le test de ces 21 modèles (20 cluster + le modèle général), le fichier **CNN_ModelExpression.py** est sur mon GitHub : `https://github.com/mateo-meynier/stageM1`
Pour voir le code du pré-traitement des données, le fichier **TL_preload.py** est disponible sur le GitHub.

Malheureusement, pour tester mon code cette fois-ci, mon ordinateur personnel n'était pas assez puissant. Il m'a fallu donc accéder au serveur distant de notre équipe qui possède la puissance nécessaire pour faire fonctionner mes scripts. J'ai dû partager le compte de Mathys car le responsable service informatique était parti en vacances donc ceci m'a ralenti dans ce projet.

Lors du test, les résultats n'étaient pas bons et aucun modèle ne semblaient bien prédire autant pour ses propres données que pour les données des autres modèles. Or Mathys avait travaillé avec le même modèle et avait eu de bon résultats.
Mon stage finissant le jour après la découverte des ses résultats, je n'ai pas pu découvrir si le problème venait de mon code ou si les données étaient différentes de celles de Mathys et ne permettaient pas une bonne prédiction. Ainsi je n'ai pas de réels résultats à présenter.

7 Conclusion

Durant mon stage, j'ai pu me plonger dans les réseaux de neurones et l'apprentissage profond, un domaine que je souhaitais aborder depuis longtemps. Même si au final, pour le projet sur lequel je travaillais une approche comme celle des réseaux de neurones à convolution était moins performante qu'un simple modèle de régression logistique, j'ai pris du plaisir à travailler dessus et j'ai surtout appris beaucoup de choses. J'ai notamment vu ce que pouvais accomplir l'apprentissage profond sur un domaine comme la génomique et j'espère voir dans le futur des projets portés sur ce domaine révolutionner le monde de la bioinformatique et de la biologie en général.

8 Bibliographie

Références

[1] N. Tsanov, A. Samacoits, R. Chouaib, A. M. Traboulsi, T. Gostan, C. Weber, C. Zimmer, K. Zibara, T. Walter, M. Peter, E. Bertrand, and F. Mueller. smiFISH and FISH-quant - a flexible single RNA detection approach with super-resolution capability. *Nucleic Acids Res.*, 44(22) :e165, 12 2016.

[2] Christophe Menichelli, Vincent Guitard, Rafael M. Martins, Sophie Lèbre, Jose-Juan Lopez-Rubio, Charles-Henri Lecellier, and Laurent Bréhélin. Identification of long regulatory elements in the genome of plasmodium falciparum and other eukaryotes. *bioRxiv*, 2020.

[3] J. Zhou and O. G. Troyanskaya. Predicting effects of noncoding variants with deep learning-based sequence model. *Nat. Methods*, 12(10) :931–934, Oct 2015.

[4] V. Agarwal and J. Shendure. Predicting mRNA Abundance Directly from Genomic Sequence Using Deep Convolutional Neural Networks. *Cell Rep*, 31(7) :107663, May 2020.

[5] D. R. Kelley, Y. A. Reshef, M. Bileschi, D. Belanger, C. Y. McLean, and J. Snoek. Sequential regulatory activity prediction across chromosomes with convolutional neural networks. *Genome Res.*, 28(5) :739–750, 05 2018.

[6] Michael Nielsen. Neural networks and deep learning.

[7] G. Eraslan, ?. Avsec, J. Gagneur, and F. J. Theis. Deep learning : new computational modelling techniques for genomics. *Nat. Rev. Genet.*, 20(7) :389–403, 07 2019.

[8] Mathys Grapotte, Manu Saraswat, Chloé Bessière, Christophe Menichelli, Jordan A. Ramilowski, Jessica Severin, Yoshihide Hayashizaki, Masayoshi Itoh, Michihira Tagami, Mitsuyoshi Murata, Miki Kojima-Ishiyama, Shohei Noma, Shuhei Noguchi, Takeya Kasukawa, Akira Hasegawa, Harukazu Suzuki, Hiromi Nishiyori-Sueki, Martin C. Frith, , Clément Chatelain, Piero Carninci, Michiel J.L. de Hoon, Wyeth W. Wasserman, Laurent Bréhélin, and Charles-Henri Lecellier. Discovery of widespread transcription initiation at microsatellites predictable by sequence-based deep neural network. *bioRxiv*, 2020.

[9] Basic cage technology.