MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE ABDELHAMID IBN BADIS - MOSTAGANEM

Faculté des Sciences Exactes et d'Informatique

Département de Mathématiques et informatique

Filière : Informatique

RAPPORT DE MINI-PROJET

Option : **Ingénierie des Systèmes d'Information**

THEME :

CLASSIFICATION AUDIO BASEE SUR L'APPRENTISSAGE PROFOND.

Etudiante : « **Legaid Raouda** »

Encadrant: « **Mohammed Elamine Moumene** »

Année Universitaire 2022-2023

Résumé

Les progrès récents dans le domaine de l'apprentissage automatique ont conduit à un intérêt croissant pour de nombreux problèmes de classification, impliquant notamment des données sous forme d'images, de vidéos et de fichiers audio. L'un des principaux problèmes de classification est celui de distinguer la catégorie d'un signal sonore. Pour cela, l'apprentissage profond est devenu l'approche la plus prometteuse pour résoudre ce genre de problèmes. Dans notre travail, nous visons à concevoir et implémenter un réseau de neurones convolutionnel dans le but de classifier des signaux sonores tels que des bruits urbains.

Mots clés : Intelligence Artificielle ; l'apprentissage profond; CNN ; classification audio.

Abstract:

Recent advances in the field of machine learning have led to a growing interest in many classification problems, especially involving data in the form of images, videos and audio files. One of the main classification problems is that of distinguishing the category of a sound signal. For this, deep learning has become the most promising approach to solve this kind of problems. In our work, we aim to design and implement a convolutional neural network in order to classify sound signals such as urban noises.

Keywords: Artificial intelligence; deep learning; CNN; audio classification.

Liste des figures

Liste des abréviations

Abréviation	Libellé complet
ML	Machine learning
DL	Deep Learning
CNN	Convolutional Neural Network
AI	Artificial intelligence
ANN	Artificial Neural Networks
RNN	Recurrent neural networks
DCNN	Deep convolutional neural network
MLP	Multi layer perceptron
SLP	Single Layer Perceptron
LSTM	Long short-term memory
MFCC	Mel-frequency cepstrum
STFT	Short-time Fourier transforms
TFCNN	Convolutional neural networks based on temporal-frequency
AAML	Additive angular margin loss
ESC	Environmental sounds classification
CQT	Chroma Constant Q-transform
DenseNet	Dense Convolutional Network
API	Application programming interface
ResNet	Residual Neural Network
CRNN	Convolutional Recurrent Neural Networks

Table des matières

Introduction Générale

A. Contexte objectif

De nos jours, il apparaît essentiel de concevoir des outils de classification automatique qui fournissent des moyens significatifs et efficaces pour décrire le contenu audio. Ça peut être la classification de bruits urbains tels que le forage, l'aboiement de chiens, le bruit de sirènes, klaxon de voiture, etc. Mais ça peut être dans plusieurs autres domaines aussi tel que le domaine médical (la classification de bruit cardiaques ou pulmonaires). Une approche qui est devenue populaire pour la classification audio est celle de l'apprentissage profond (Deep Learning).

Le Deep Learning est une technique d'apprentissage permettant à un programme, par exemple, de reconnaître le contenu d'un fichier audio ou de comprendre le type bruit « La technologie du Deep Learning apprend à modéliser le monde. C'est-à-dire comment la machine va représenter le son ou l'image par exemple ». Ce système d'apprentissage et de classification, basé sur des « réseaux de neurones artificiels » numériques est une technique courante en IA, permettant aux machines d'apprendre et reconnaitre des objets.

Dans notre projet, nous visons à étudier l'utilisation des réseaux de neurones convolutifs(CNN) pour la classification de données sonores, notamment les bruits urbains.

B. Organisation du mémoire

Notre travail est divisé en deux parties, dans la première partie nous faisons un état de l'art sur les méthodes de classification de sons afin d'avoir une vision générale. Le deuxième chapitre est consacré à l'introduction du Deep Learning, une description plus détaillée sur les réseaux de neurones convolutifs (CNNs) qui est la méthode choisie dans notre projet.

CHAPITRE1

Classification de bruits urbains

Chapitre 1

Classification de bruits urbains

1. Introduction

Cette classification du son urbain est un aspect important de diverses applications émergentes, telles que la surveillance, la compréhension du paysage sonore urbain et l'identification des sources de bruit, c'est pourquoi le sujet de recherche a beaucoup retenu l'attention ces dernières années. L'objectif de ce petit projet est de développer un système efficace basé sur l'apprentissage automatique pour la classification des sons urbains dans des conditions de bruit réelles [1].

L'une des principales exigences concerne la caractérisation du son urbain, qui englobe plusieurs tâches, telles que la classification et la segmentation des sons [2]. On estime que les grandes villes doivent gérer des milliers d'événements concomitants. Parmi les sons environnementaux, il existe plusieurs catégories, telles que les sons naturels et artificiels, comme les gens qui parlent et la musique, et les bruits anormaux, tels que les bruits de coups de feu. Les sons urbains souvent associés à des bruits d'origine humaine sans rapport avec la parole ou l'interprétation de la musique sont des bruits urbains quotidiens tels que les klaxons et les sirènes, les aboiements de chiens et autres, qui peuvent être classés comme anormaux du point de vue des sons urbains et peuvent nécessiter une attention particulière.

Quant à la classification sonore urbaine, qui a pour objectif principal la détection d'événements sonores pertinents acquis à partir de scénarios urbains, la plupart des solutions proposées sont basées sur des réseaux de neurones convolutifs (CNN). Certains travaux sont également basés sur les réseaux de neurones récurrents (RNN).

La Figure1 présente l'évolution du nombre d'articles publiés par année pour le traitement ou la classification des sons environnementaux et urbains. De cette figure, il est possible de déduire que la croissance du Deep Learning (DL) a suscité un intérêt croissant dans la communauté scientifique de ce domaine. Par conséquent, les approches proposées les plus récentes sont basées sur des méthodes de Deep Learning (DL) [3].

2. Méthodes de classification sonore

Les modèles statistiques de signaux audio reposent en général sur des descripteurs audio. Les modèles couramment utilisés dans la classification audio sont les mélanges de gaussiennes, les machines à vecteur de support, les modèles de Markov cachés ou les réseaux de neurones convolutifs profonds. De bonnes méthodes de classification peuvent être appliquées dans plusieurs domaines, y compris la surveillance, l'atténuation du bruit et l'informatique contextuelle. Par conséquent, pour la plupart attribuer avec précision une classe à un son spécifique, plusieurs modèles de Machine Learning (ML) ont été développé pour extraire les caractéristiques nucléaires des échantillons audio étudiés pendant la formation, puis classer les audio invisibles avec une grande confiance.

2.1 Les réseaux de neurones

Les chercheurs ont identifié certaines limites qui les empêchaient d'obtenir des bons résultats sur les tâches de classification sonore. Par conséquent, utilisé un réseau de neurone convolution profond (DCNN) en combinaison avec des données augmentant techniques, à savoir l'injection de bruit, le temps de changement de vitesse et le changement de hauteur et de vitesse, parmi les ensemble de formation pour résoudre la rareté des données étiquetées. En revanche, En utilise une Mémoire à long court terme (LSTM) en combinaison avec des caractéristiques spectrales obtenues à partir de segment de formation audio. Et exploré l'utilisation d'un modèle de réseau neuronal convolutif (CNN). Avec une perte de marge angulaire additive spécifique (AAML) et plus couramment utilisé empilées caractéristiques, coefficients Cepstre à fréquence Mel (MFCC) et chroma gramme en combinaison avec un réseau de neurones convolutifs (CNN).

On va utiliser un Audio Modèle de réseau prototype (Apnée) composé d'un auto-encodeur et d'un classifier. On introduit un modèle basé sur le réseau de neurones convolutifs (CNN) associé avec des mécanismes d'attention, appelés Réseaux de neurones convolutifs basés sur l'attention temporelle-fréquence (TFCNN).

Il existe trois étapes fondamentales pour effectuer une classification sonore : le prétraitement, l'extraction des caractéristiques et la classification, comme le montre la figure1. Par conséquent, cette revue de la littérature présente les travaux les plus prometteurs pour aborder chaque étape[4]

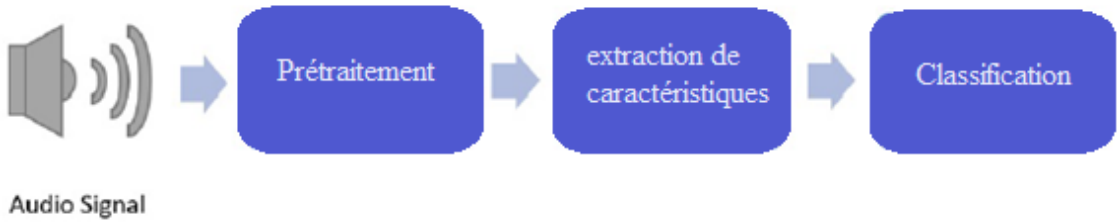


Figure1 : Méthodes de classification de signal sonore.

L'objectif des modèles est d'avoir de bonnes performances de généralisation pour les données invisibles, ce qui nécessite généralement de grandes quantités de données pour entraîner efficacement les modèles.
Pour faire face à la rareté des données étiquetées pour la classification des sons environnementaux (ESC), nous proposons quatre techniques d'augmentation différentes à appliquer à l'ensemble d'apprentissage d'origine :
• Étirement temporel : ralentit ou accélère les échantillons audio, mais la hauteur reste inchangée.
• Changement de hauteur : la hauteur des échantillons audio est augmentée ou diminuée tout en gardant la durée inchangée.
• Compression de plage dynamique : compressez la plage dynamique de l'audio à l'aide des paramétrages de la norme Dolby E et du serveur de diffusion radio en ligne Icecast.
• Ajout de bruit de fond : mélangez les enregistrements des sons de fond de différentes scènes avec les échantillons audio.
Une analyse détaillée des différentes techniques est effectuée pour déterminer l'impact des différentes techniques d'augmentation de données sur la précision finale, permettant la quantification des contributions de chacune des transformations de données employées sur les données d'apprentissage, suggérant qu'une classe-conditionnelle technique d'augmentation pendant la formation serait bénéfique.

Les caractéristiques et les modèles qui peuvent atteindre une meilleure précision. Das et al. [5] ont présenté une étude comparative entre un réseau de neurones convolutifs (CNN) et un modèle de mémoire à long-court terme (LSTM) utilisant différentes combinaisons de caractéristiques spectrales. Tout d'abord, le signal audio a été prétraité pour réduire la quantité d'informations redondantes ; le théorème de Nyquist – Shannon stipule que les taux d'échantillonnage doivent être au moins deux fois
la valeur de la fréquence d'une forme d'onde continue. Cependant, pour réduire le temps de formation, le sous-échantillonnage a été réalisé en utilisant le taux d'échantillonnage par défaut de la bibliothèque librosa de 22 050 Hz. L'étape suivante correspond à l'extraction de caractéristiques spectrales telles que les coefficients Cepstre de fréquence Mel (MFCC),
Mel spectrogramme, Chroma Short-Term Fourier Transformation (STFT), Chroma Constant Q-transform (CQT), Chroma Energy Normalized Statistics (CENS), Spectral Contrast et Tonnetz. Les caractéristiques spectrales extraites, combinées aux techniques d'augmentation de données de pitch shift, time stretch et pitch shift avec time stretch, avec les modèles finaux utilisés dans la classification des événements sonores et avec une évaluation détaillée de la précision respective, ont conduit aux conclusions suivantes :

• Une augmentation du nombre d'époques a entraîné une diminution exponentielle de l'erreur de validation des données d'apprentissage et de test.
• Le modèle de mémoire à long court terme (LSTM) avait de meilleures performances, dans la plupart des cas, que le réseau neuronal convolutif (CNN), ce qui devient plus notable avec les techniques d'augmentation des données, car la cellule de mémoire à long court terme (LSTM) englobe la rétro propagation d'erreur constante, ce qui permet de mieux traiter les données bruitées.
• En se concentrant sur l'influence des différentes caractéristiques utilisées, celle qui a conduit à la meilleure précision était les coefficients cepstre de fréquence Mel (MFCC) ; cependant, il a été possible de surpasser la précision obtenue en utilisant une pile de caractéristiques différentes, principalement des coefficients cepstre de fréquence Mel (MFCC) et la transformation de Fourier à court terme Chroma (STFT).

3. Conclusion

Une meilleure compréhension des sons urbains pourrait grandement contribuer à améliorer l'habitabilité urbaine, comme un contrôle du bruit plus efficace, une surveillance de la sécurité fiable et une meilleure planification de l'environnement acoustique. Cet article se concentre sur la récupération de contenu sonore urbain qui est considérée comme un élément essentiel des villes intelligentes. la classification du son comporte trois étapes principales : le prétraitement de l'audio d'entrée signal, l'extraction de caractéristiques acoustiques à partir du signal audio prétraité et du signal audio classification [4].

Par conséquent, les chercheurs ont proposé plusieurs modèles pour relever ce défi [9], en se concentrant sur différentes étapes de la tâche. Certains essaient de développer ou d'améliorer l'architecture du modèle en utilisant des versions modifiées des fonctions de perte, des méthodes pour supprimer des parties de la séquence d'entrée, ou en explorant divers types d'architectures telles que Deep Convolutional Neural Network (DCNN), Convolutional Recurrent Neural Networks (CRNN) , Long-Short Term Memory (LSTM), Residual Neural Network (ResNet), Dense Convolutional Network (DenseNet) et plus récemment Transformers. La plupart des approches proposées étaient basées sur des méthodes de Deep Learning (DL).

CHAPITRE 2

Réseaux de neurones

Chapitre2
Réseaux de neurones

8. Introduction

Nous avons mentionné dans le chapitre 1 que l'intelligence artificielle comportait diverses branches, celles-ci consistent en plusieurs aspects d'autonomie de la machine ou plus exactement, un phénomène dont l'apprentissage devient autonome. Parmi ces branches, nous avons celle du Machine Learning qui veut dire apprentissage automatique de la machine.

Ce même domaine peut lui aussi se diviser en tant qu'apprentissage supervisé, non supervisé, ou aussi apprentissage par renforcement. En vue des développements qui ont suivi cette discipline, les recherches ont rapporté un autre aspect de l'apprentissage qui a vu le jour dans les années après (presque 30 ans après l'apparition du Machine Learning) appelé, apprentissage profond ou Deep Learning. La Figure2 illustre la position du Deep Learning au sein du domaine de l'intelligence artificielle aussi dénommé AI (pour Artificial Intelligence). Nous allons donc dans ce chapitre détailler les principes de base d'un réseau profond et ses principales caractéristiques qui font son importance dans le domaine du traitement audio [4].

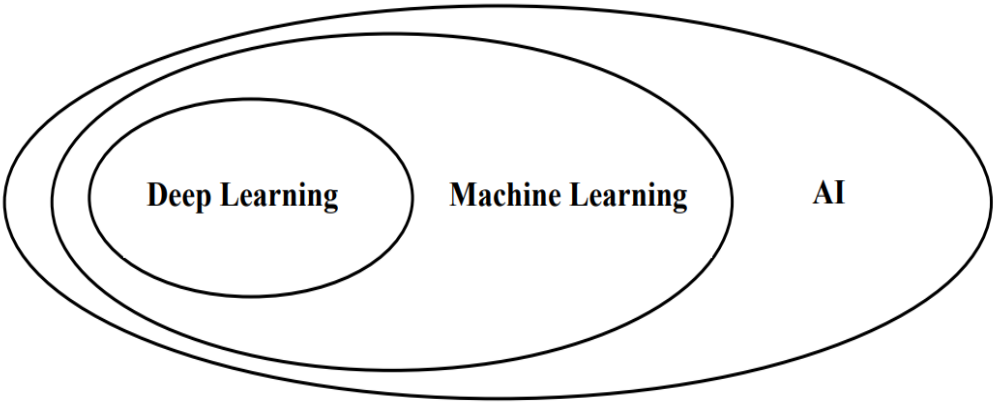


Figure2 : Hiérarchie et position du DL par rapport au Machine Learning et IA.

9. Réseaux de Neurones Artificiels(ANN)

Avant de comprendre ce qu'est un réseau de neurones (Neural Network) artificiel, il faut tout d'abord comprendre ce qu'est le Machine Learning, grâce à quelques généralités toutes simples sans détailler étant donné que c'est le Deep Learning qui nous intéresse dans notre travail, nous pourrons ainsi comprendre la suite qui concerne le réseau de neurones ainsi que le Deep Learning.

Le Machine Learning peut être caractérisé par un modèle mathématique qui suit des données d'apprentissage, on obtient ce dernier par des entrées de données X et des sorties cibles Y pour le cas d'un apprentissage supervisé (ce qui n'est pas toujours le cas). À partir de là, la machine obtient plus d'expérience en se familiarisant avec ces données appelées « Dataset »[5] , à partir de ces données Dataset(X , Y) où X représente les caractéristiques nommées « Features » et Y la cible « Target », leur importance réside en tant qu'information utile qui sera utilisée par le modèle via le concept de « Training » qui lui permettra donc par le biais de cet entrainement de tracer son modèle qui va pouvoir représenter le processus d'apprentissage automatique . Plus les

caractéristiques concernées sont bien définies plus le modèle sera précis [5]. La figure suivante montre un exemple de Machine Learning qui va avoir en résultat un modèle :

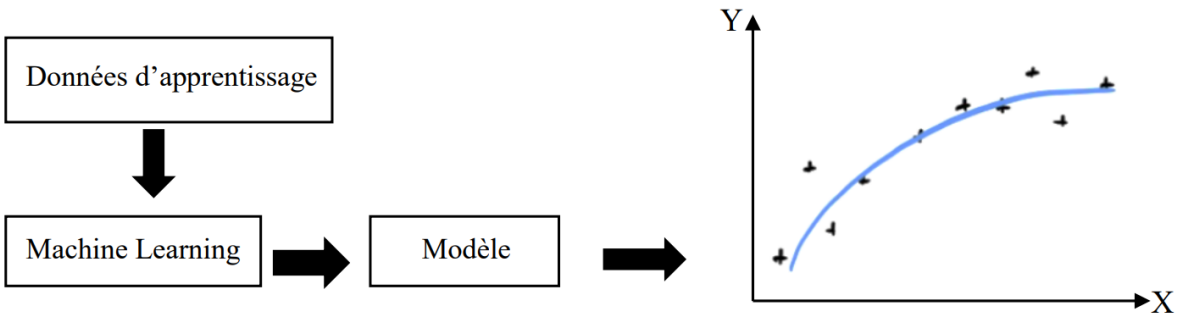


Figure3: Ce qui se passe pendant le processus d'apprentissage.

Donc comme cité, les données d'apprentissage vont influencer sur la valeur de y par le facteur feature qui va résulter d'un modèle comme illustré sur la Figure3 en bleu où l'espace qu'il y'a entre le modèle en bleu avec les points dans l'espace représente la précision (Accuracy) ou l'erreur entre notre modèle et les données. Ceci constitue donc une fonction de prédiction $f(X)$ pour minimiser les erreurs via un algorithme d'optimisation [6].

Ce type de calcul pour la machine est très efficace pour prédire des calculs automatiques pour les statistiques ou la science de données en général, mais supposant que nous avons une large quantité de données comme des centaines de milliers de sons à faire entrer avec de grosses tailles, on aura donc une très large quantité de pixels, ce qui rend le calcul trop grand voir même limité en machine Learning. C'est pourquoi vient alors la notion de réseau profond afin de traiter ce genre de données et de l'exploiter dans le Deep Learning [7]

Nous pouvons ainsi traduire cette notion dans la Figure4 suivante comme suit :

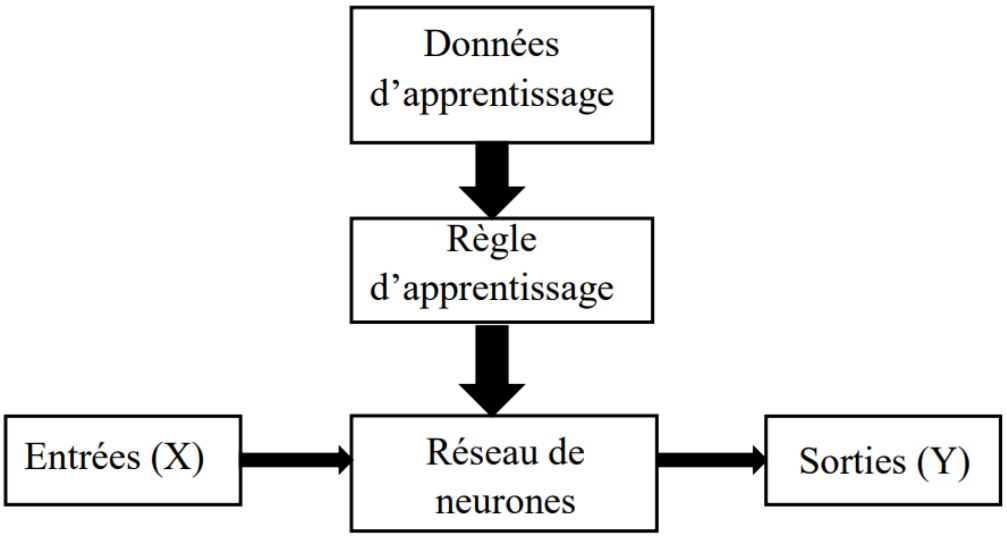


Figure4: Relation entre Machine Learning et Deep Learning.

De là, notre modèle en Machine Learning est remplacé par un réseau de neurones et la règle d'apprentissage prend place au Machine Learning. Dans ce contexte de réseau de neurones, la règle d'apprentissage fera office de processus à déterminer le modèle (réseau de neurones) . Il

faut aussi faire la différence entre les données d'apprentissage qui sont les données sur qui le modèle s'entraîne, et les données d'entrées qui seront les données à prédire en sortie.

2.1 Perceptron

Le perceptron est en quelque sorte le niveau le plus bas et le plus simple à réaliser qu'un réseau de neurones artificiel puisse avoir [9]. Ceci vient du fait qu'un neurone artificiel est tiré d'un neurone naturel en faisant le rapprochement comme le montre la Figure5 suivante :

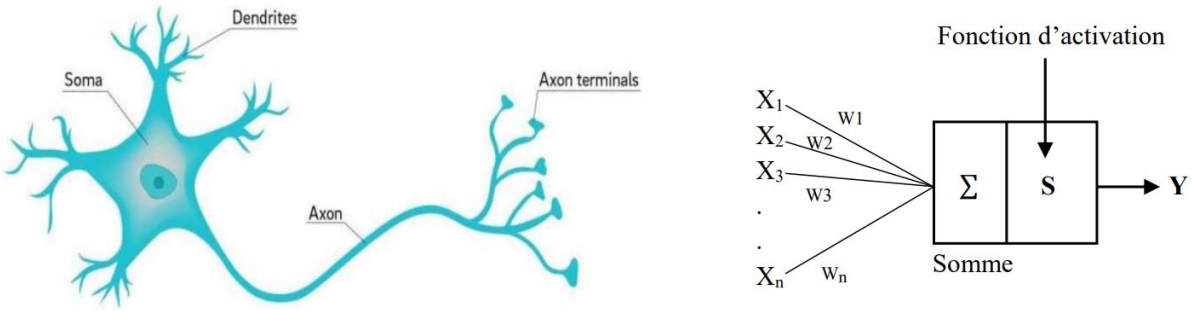


Figure5 : a. Neurone biologique **b.** Neurone artificiel

Un neurone biologique fait passer des informations au travers des synapses[8]dans la dendrite, vers d'autres neurones biologiques, le même principe est adopté pour un neurone artificiel qui fera office d'entrés de données X vers la sortie Y (Axonterminals); cette transmission d'information est provoquée par une excitation ou inhibition grâce aux stimulus qu'il reçoit, en faisant l'analogie avec un neurone artificiel, cela se traduit par une fonction de transfert, aussi appelée fonction d'activation[5]. La transformation appliquée par la fonction d'activation de la donnée se fait par seuillage (Thresholding) afin de déterminer une fonction de décision [9].

En regardant bien dans la Figure6 sur le neurone artificiel on a une combinaison linéaire qui s'opère tel que :

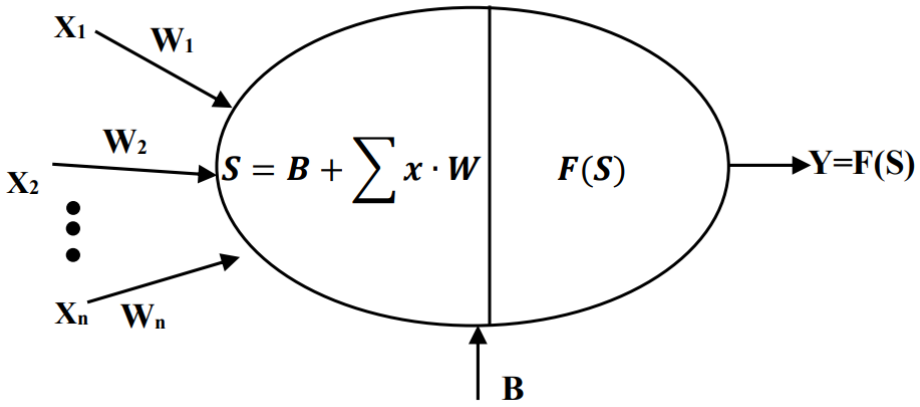


Figure6 : Relation de linéarité dans un Perceptron.

De là commence le principe d'apprentissage de base grâce aux poids synaptiques $\boldsymbol{Wn}$ et le biais $\boldsymbol{B}$ entrant en phase d'agrégation et en phase d'activation (Threshold). Chacun de ces deux influent sur le comportement du perceptron et donc le modèle en sortie[10], soit par un offset par

le biais ou changement de magnitude voir même le signe par les poids synaptiques, étant donné que nous avons une sorte de linéarité qui ressemble à une droite $\boldsymbol{F(X) = aX + B}$. Cette influence sur le comportement du modèle pour arriver à la sortie désirée se fait par une mise à jour des paramètres (Weights update), c'est pourquoi les paramètres $\boldsymbol{Wn}$ et $\boldsymbol{B}$ sont des paramètres qui entrent dans l'entraînement de la machine ou plus précisément l'algorithme d'apprentissage qui suit l'équation suivante :

$$\mathbf{W = W + \alpha\,(Y_{ref} - Y)\,X}$$

À travers ce genre de mise à jour que se fait une sorte de réglage automatique via cet algorithme qui permet de renforcer les paramètres des poids synaptiques à chaque fois qu'une entrée X est activée en même temps que la sortie Y présente dans ces données[5]. Ceci dit, cela reste très limité en termes de ce que doit apporter le domaine du réseau de neurones, c'est pourquoi les fonctions d'activation aident à avoir un modèle qui entre dans la non-linéarité, nous avons par exemple des fonctions très connues comme la fonction sigmoïde par exemple définie tel que :

$$F(S) = \frac{1}{1+e^{-S}} \qquad (1.4)$$

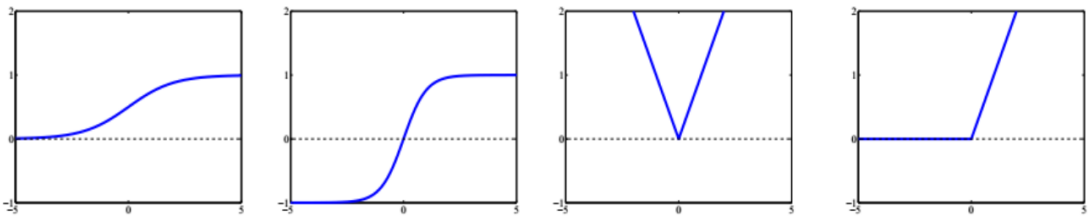


Figure7 : Exemple de fonctions non linéaires appliquées dans un Perceptron.

Selon la Figure7 nous pouvons citer les fonctions en partant de gauche, la fonction logistique (Sigmoïde), la fonction tangente hyperbolique, la fonction valeur absolue, et la fonction Unité Linéaire Rectifiée (ReLU pour Rectified Linear Unit) qu'on aura l'occasion de voir dans le prochain chapitre qui sera utilisé dans notre travail.

$$\mathbf{Tanh(S) = \frac{1-e^{-2(s)}}{1+e^{-2(s)}}}$$

$$\mathbf{Abs(S) = |S|}$$

$$\mathbf{F = max\,(0,\,S)}$$

Avec bien sûr, S qui représente les (weightedsum) c'est-à-dire la somme des produits entre la matrice des poids synaptiques W et les entrées du réseau en ajoutant aussi le vecteur biais B. On utilise les matrices pour simplifier les calculs mathématiques.

2.1.1 Perceptron Multi Couche (Multi Layer Perceptron MLP)

Il contient qu'une partie classification, Cette contrainte de modèle linéaire du perceptron comme le problème logique XOR sur l'impossibilité séparation linéaire[11], qui le rend peu utile pour des phénomènes de vie courante qui sont non-linéaires, il peut être résolu par le développement du Perceptron multi couche, en connectant plusieurs neurones ensemble, il est possible de résoudre des problèmes plus complexes. Nous pouvons pour le cas d'un perceptron l'adapter avec plusieurs perceptrons en une seule couche (Figure8), par exemple, pour classifier des audio de multiples classes différentes en sorties. Pour cela en connectant les entrées avec toutes les

sorties et que chaque perceptron de cette couche de sortie ait son propre biais ***B***[12]. La figure suivante montre la différence entre un réseau de perceptrons mono couche, et un réseau de neurones multi couches

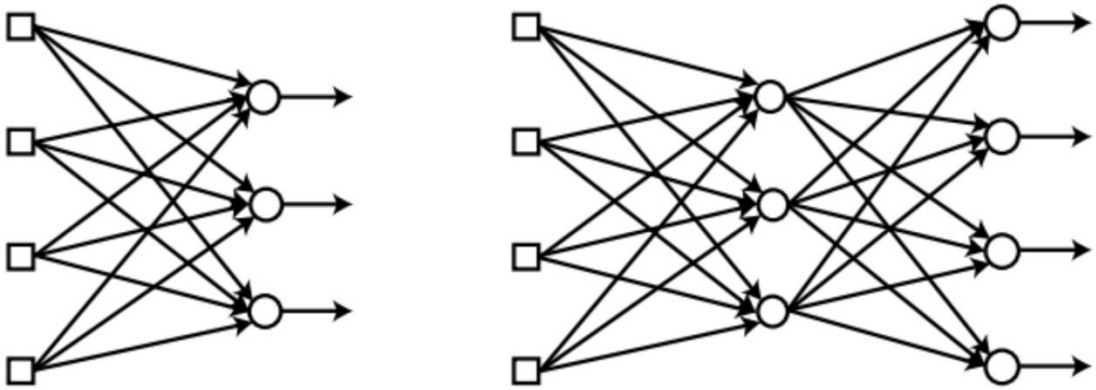


Figure8 : Différence entre réseau de neurones à une seule couche et multicouche.

Dans le cas d'un SLP (Single Layer Perceptron à gauche de la figure) on peut avoir un seul perceptron comme on peut avoir plusieurs perceptrons dans une seule et unique couche [12], c'est pourquoi on fait la différence entre SLP et MLP (Multi Layer Perceptron à droite de la figure).

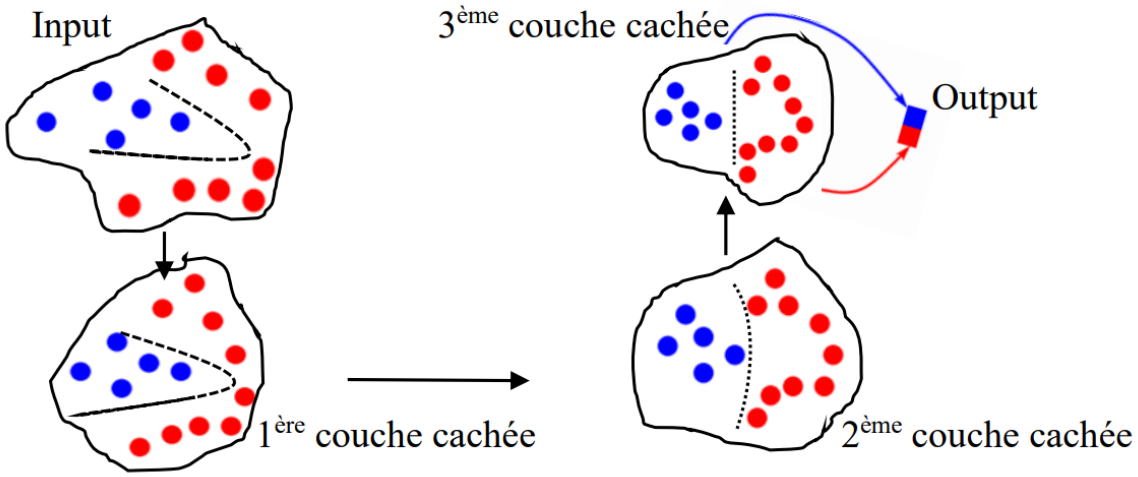


Figure9 : Projection des données dans un réseau multi couches.

Afin de mieux comprendre le réseau de perceptrons Multicouches, la Figure9 illustre ce qui se passe lorsque les données traversent plusieurs couches qui sont traitées de manière non linéaire en fonction des caractéristiques extraites des données par l'algorithme d'apprentissage au fur et à mesure que celles-ci se rapproche d'une couche à une autre vers la sortie qui devient linéaire [12]. À partir de cette idée, nous pouvons maintenant comprendre ce qu'est la profondeur d'un réseau de neurones.

En insérant des couches intermédiaires entre la couche d'entrée et de sortie (Figure 10) nommées Hidden layers en français couches cachées ; nous avons chaque neurone reçoit à son entrée les sorties des neurones de la couche précédente ; en d'autres termes, la couche de sortie reçoit en entrée les sorties de la dernière couche cachée, et celle-ci reçoit à son tour les sorties de la couche d'entrée précédente; c'est ce qu'on appelle réseau de propagation avant (Feed Forward

network), c'est à dire que nous n'avons pas de retour d'une couche vers la couche qui l'a précède, l'information des données démarre depuis la couche d'entrée vers la couche de sortie par l'intermédiaire des couches Hidden layers. Lorsque le nombre de couches cachées augmente alors ici on parle de Deep Learning car c'est à travers cette profondeur que l'exploitation des caractéristiques des données devient intéressante, c'est pourquoi les branches d'un réseau de neurones dépendent de l'architecture des couches [13].

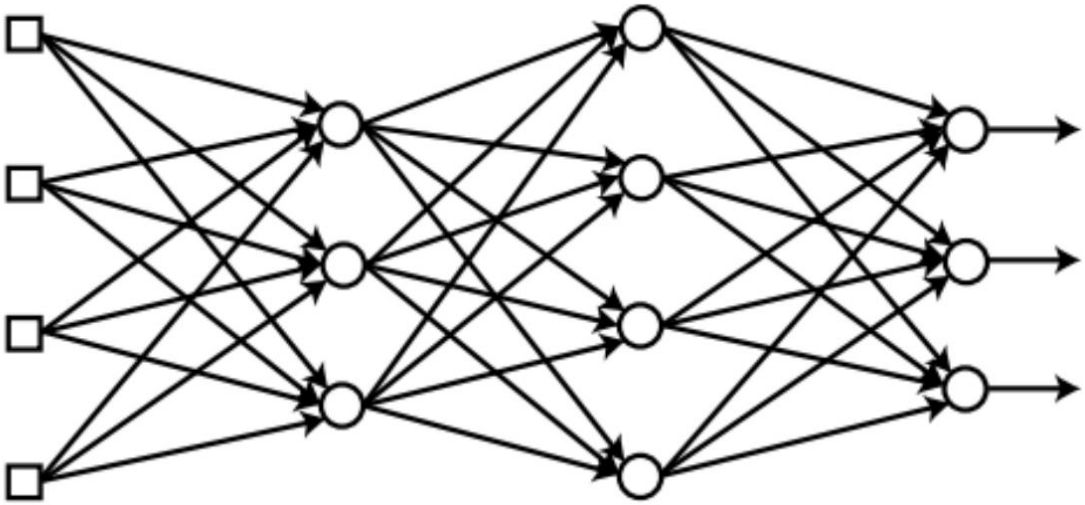


Figure10 : Réseau de neurones profond [5].

2.1.2 Les architecture des réseaux de neurones

Il existe des architectures qui sont très utilisés en Deep learning, parmi les plus populaires, nous avons l'algorithme de réseau de neurones convolutif CNN (Convolutional Neural Network) et le réseau de neurones récurent RNN (Recurrent Neural Network). Comme le montre la Figure11 :

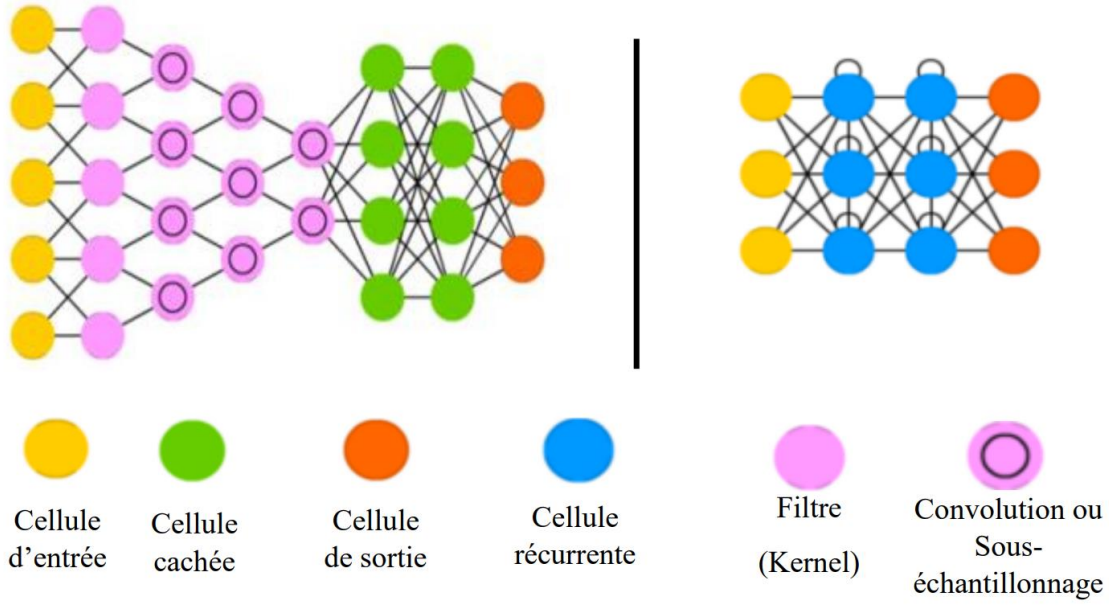


Figure11 : Architecture générale d'un réseau CNN (à gauche) et RNN (à droite)[14].

À partir de cette figure on peut dire qu'en fonction des besoins désirés, on peut implémenter une architecture à réseau de neurones comme dans notre cas par exemple, nous nous intéressons pour l'architecture CNN, car ils présentent beaucoup de performances surtout pour le cas de big data.

2.2 Propagation vers l'avant (Forward propagation)

L'apprentissage en profondeur et les réseaux de neurones ont entamé une révolution, Et la première étape de la formation d'un réseau de neurones est la propagation vers l'avant.

Maintenant, si vous remarquez que nous avons soigneusement calculé les activations cachées pour chacun des neurones de manière séquentielle, c'est-à-dire l'une après l'autre. En fait, une optimisation que l'on peut apporter au calcul est de les calculer en parallèle. Étant donné que le calcul des neurones est indépendant les uns des autres, ils peuvent être calculés en parallèle facilement. Nous pouvons calculer simultanément la première activation cachée.

C'est la série de calculs qui nous emmène de l'entrée à la sortie. Comme on peut le voir dans la (Figure12).

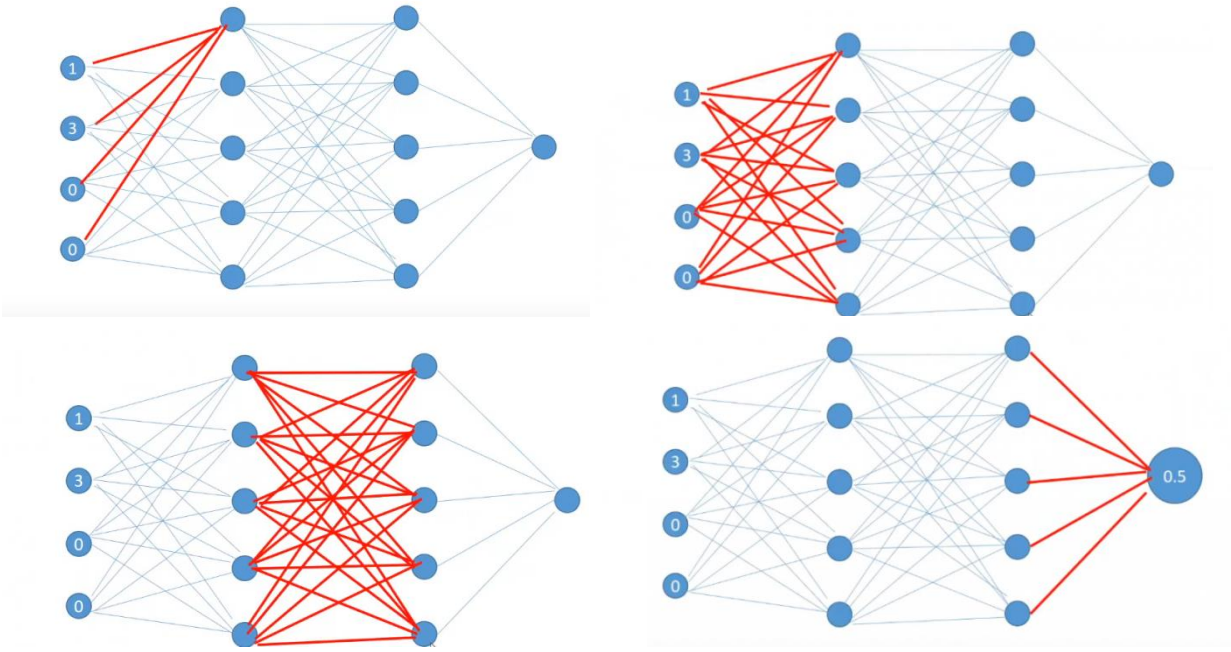


Figure12 : Les étapes de calcul sur la Propagation vers l'avant

2.2.1 Erreurs dans le réseau neuronal

Jusqu'à présent, nous avons vu comment la propagation vers l'avant nous aide à calculer les sorties. Disons que pour une ligne particulière, la cible réelle est 0 et la cible prédite est 0,5. Nous pouvons utiliser cette valeur prédite pour calculer l'erreur pour une ligne particulière. Le type d'erreur que nous avons choisi ici est Erreur au carré (Squared Error).

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y})^2$$

Il peut y avoir d'autres formules de calcul d'erreur mais par souci de simplicité, nous avons choisi celle-ci.

Nous savons que l'erreur dépend directement de deux quantités : les valeurs réelles et les valeurs prédites.

Ce qui peut être changé pour réduire l'erreur est l'activation cachée de la deuxième couche, et la fonction d'activation présente à la couche de sortie .Nous pouvons modifier à la fois les fonctions d'activation et les poids, mais nous ne pouvons pas modifier les fonctions d'activation de manière itérative car cela modifierait les distributions pendant le temps d'entraînement. La fonction d'activation est donc également hors de question. Il est intéressant de noter que nous pouvons utiliser une stratégie intelligente pour ajuster les pondérations et réduire les erreurs.

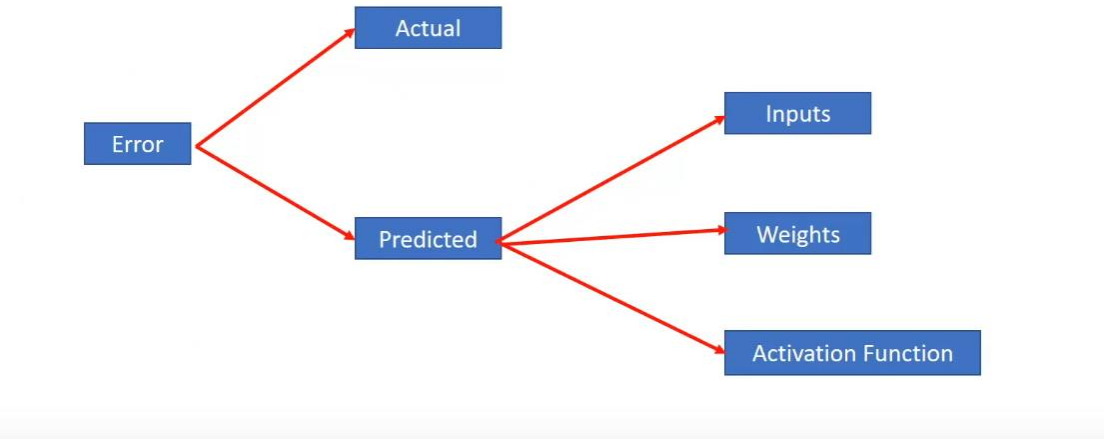


Figure13 : Les deux quantités d'Erreur les valeurs réelles et les valeurs prédites.

2.2.2 Fonction de perte (Loss Function)

La fonction de perte est très importante dans le machine learning ou le deep learning. Disons que vous travaillez sur n'importe quel problème et que vous avez formé un modèle d'apprentissage automatique sur l'ensemble de données et que vous êtes prêt à le présenter à votre client. Cette fonction vous permet donc d'être sûr que ce modèle donnera le résultat optimal, et c'est aussi une métrique ou une technique qui vous aidera à évaluer rapidement votre modèle sur le jeu de données.

En termes simples, la fonction de perte est une méthode d'évaluation de la qualité de la modélisation d'un ensemble de données par un algorithme.

Dans la régression linéaire simple, la prédiction est calculée à l'aide de la pente (m) et de l'ordonnée à l'origine (b). La fonction de perte pour cela est le (Yi - Yihat) ^ 2, c'est-à-dire que la fonction de perte est la fonction de la pente et de l'interception. Comme en peut suivre dans la Figure14 suivante :

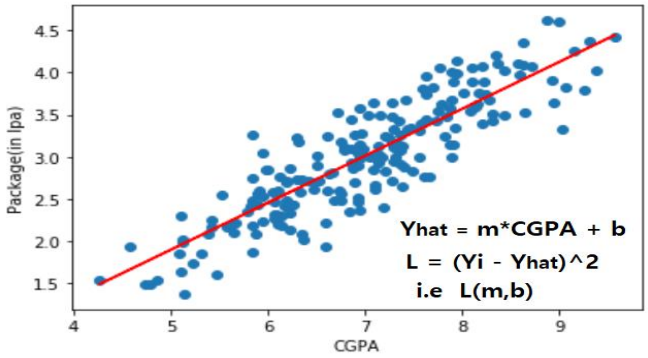


Figure 14 : La fonction de perte dans la régression linéaire simple.

Perte de régression

1. Erreur quadratique moyenne/perte quadratique/perte L2 :

$$\mathbf{MSE} = \frac{1}{n}\sum_{i}^{N}(Y_i - \hat{Y})^2$$

1. Erreur absolue moyenne/perte L1 :

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{N}|Y_i - \hat{Y}_i|^2$$

2.2.3 Fonction Coût (Cost Function)

Nous avons introduit l'erreur entre le modèle et les données. L'ensemble de ces erreurs est représenté par une fonction coût (En anglais Cost ou loss function). Cette fonction peut être définie avec plusieurs fonctions, l'une des plus fréquemment utilisée est l'erreur quadratique moyenne (Mean squared error), tel que :

$$\mathbf{E} = \frac{1}{2m}\sum_{i=1}^{m}(\mathbf{Y_{ref}} - Y^2)$$

Elle se traduit par la moyenne des erreurs $\boldsymbol{m}$ du nombre de neurones en sortie. $Yref$ change à chaque fois qu'il y'a une nouvelle mise à jour des paramètres. En réseau de neurones, la fonction coût est exprimée par la technique de Back propagation [15].

2.2.4 Rétro propagation (Back-Propagation)

En réseau de neurones, nous avons détaillé le principe de Feed-Forward la propagation avant, mais on peut se poser la question sur comment la mise à jour des paramètres W et B se fait dans le réseau. Une idée consiste à faire le chemin inverse, c'est-à-dire qu'après avoir obtenu la sortie Y avec une erreur, on calcule une chaîne de gradient à partir de la sortie par rapport à la dernière couche qui précède chacun ainsi de suite jusqu'à la première couche, ce principe est illustré dans la figure suivante où nous observons l'erreur en sortie qui se propage dans la couche qui précède la sortie selon comment elle varie. On obtient donc de nouveaux paramètres (W, B) mis à jour, avec $\boldsymbol{\varphi'}$ est la variation dans chaque neurone et υ est le produit de somme des paramètres dans le réseau.

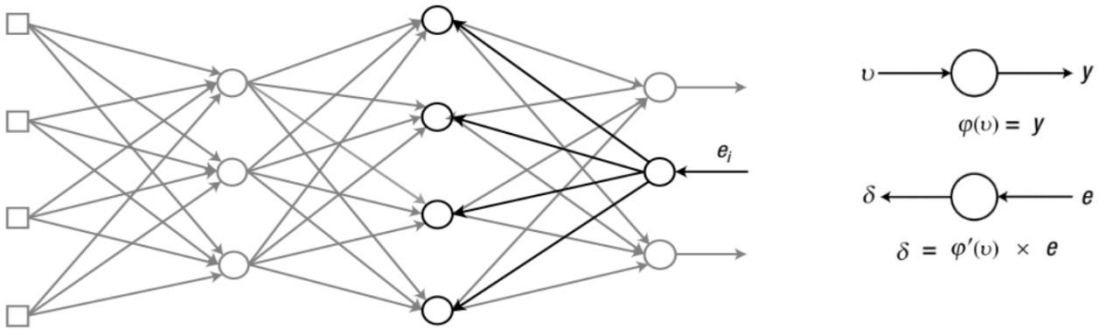


Figure15 : Illustration du principe de back propagation.

2.3 La descente de Gradient (Gradient descent)

La descente de gradient représente l'algorithme d'apprentissage optimal, elle minimise la fonction coût pour prendre le résultat optimal possible et donc elle définit le processus d'apprentissage qui optimise les poids synaptiques et réduit le coût de la fonction [7]. La descente de gradient correspond où le processus avance par l'intermédiaire d'un pas d'apprentissage, Ce sera un facteur pour l'avancement de la dérivée sur la fonction coût.

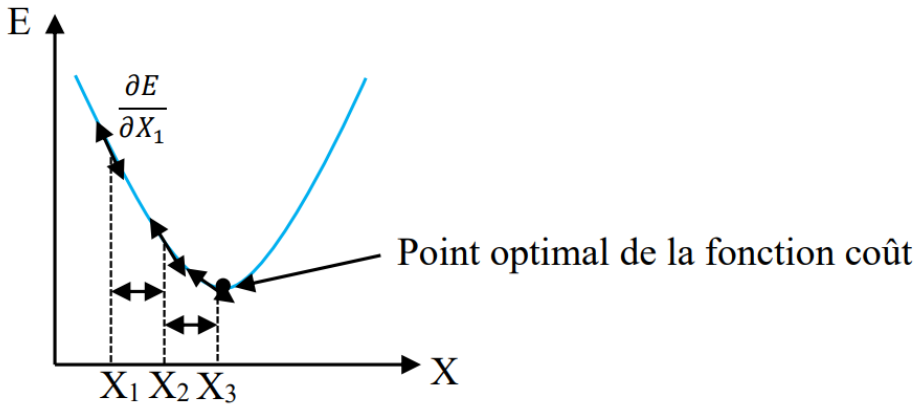


Figure16 : Optimisation de la fonction coût par descente de gradient.

Comme il est illustré sur cette figure ci-dessus, suivant des dérivées partielles qui correspondent à chaque erreur donc, l'ensemble de ces erreurs et par l'intermédiaire de la descente de gradient suivant un pas, nous avançons vers le point optimal, et si le pas et trop grand nous risquons de le dépasser, et s'il est trop petit, nous risquons de ne jamais se rapprocher de ce point.

2.4 Taux d'apprentissage (Learning Rate)

Comme il a été cité dans la section précédente, c'est le pas d'apprentissage, il correspond dans l'équation au coefficient $\boldsymbol{\alpha}$, il peut aussi exprimer la vitesse à laquelle la machine peut apprendre, cette vitesse influe sur le calcul des gradients pour optimiser la fonction coût.

2.5Sur-ajustement (Overfitting)

Cependant, il faut faire attention au sur-apprentissage (**Overfitting**) qui peut engendrer des dysfonctionnements avec des résultats de prédiction non corrects, car le fait de sur-apprendre conduit à mesurer aussi et modéliser le bruit contenu dans les données (problème de divergence vers le point optimal). Etant donné que notre étude est d'enlever le bruit des sons, il est donc clair qu'il faut éviter ce phénomène de sur-ajustement.

2.6 Regularization

Lors de la formation d'un modèle d'apprentissage automatique, le modèle peut être facilement suréquipé ou sous-équipé. Pour éviter cela, nous utilisons la régularisation dans l'apprentissage automatique pour adapter correctement le modèle à notre ensemble de test. La régularisation fait référence aux techniques utilisées pour calibrer les modèles d'apprentissage automatique afin de minimiser la fonction de perte ajustée et d'éviter le sur-ajustement ou le sous-ajustement.

Comprendre le sur-ajustement et le sous-ajustement :

Pour former notre modèle d'apprentissage automatique, nous lui fournissons des données à partir desquelles apprendre. Le processus consistant à tracer une série de points de données et à tracer une ligne de meilleur ajustement pour comprendre la relation entre les variables s'appelle l'ajustement des données. Notre modèle est le mieux adapté lorsqu'il peut trouver tous les modèles nécessaires dans nos données et éviter les points de données aléatoires et les modèles inutiles appelés bruit.

Si nous permettons à notre modèle d'apprentissage automatique d'examiner les données trop souvent, il trouvera de nombreux modèles dans nos données, dont certains sont inutiles. Il apprendra bien sur l'ensemble de données de test et s'adaptera très bien. Il apprendra des modèles importants, mais il apprendra également du bruit dans nos données et ne sera pas en mesure de faire des prédictions sur d'autres ensembles de données.

Un scénario dans lequel un modèle d'apprentissage automatique essaie d'apprendre des détails ainsi que du bruit dans les données et essaie d'adapter chaque point de données à une courbe est appelé sur-ajustement.

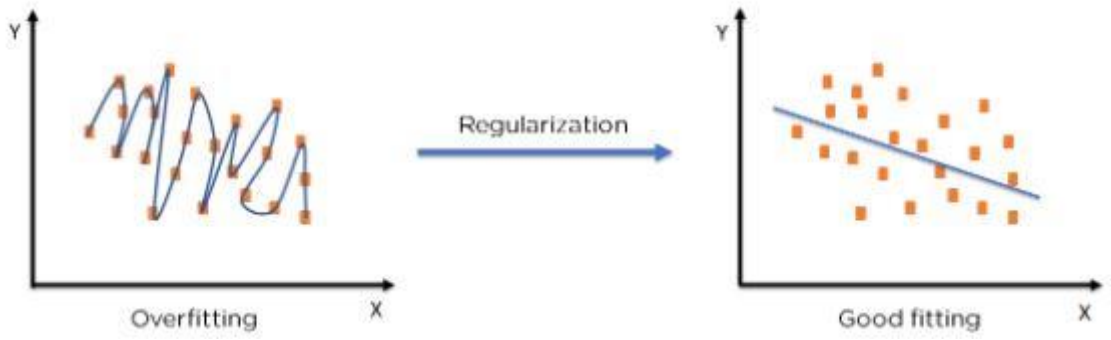


Figure17 : Le rôle de la Régularisation sur les modèles d'apprentissage automatique.

3. Réseaux de Neurones Convolutifs CNN

Les couches cachées ont inauguré une nouvelle ère, les anciennes techniques étant inefficaces, en particulier lorsqu'il s'agit de problèmes tels que la reconnaissance vocale, la reconnaissance de formes, la détection d'objets, la segmentation d'images et d'autres problèmes liés au traitement d'images. CNN est l'un des réseaux profonds les plus déployés [17]

Les réseaux de neurones convolutifs (CNN) sont devenus l'architecture de réseau de neurones dominante pour résoudre de nombreuses tâches. Même si les unités de traitement graphique sont le plus souvent utilisées dans la formation et le déploiement de CNN [18].

Les réseaux de neurones convolutifs ont une méthodologie similaire à celle des méthodes traditionnelles d'apprentissage supervisé : ils reçoivent des sons en entrée, détectent les features de chacune d'entre elles, puis entraînent un classifieur dessus.

3.1 Convolution

La convolution est une opération mathématique où les éléments du filtre sont multipliés élément par élément avec l'entrée sur laquelle le filtre est actuellement présent et les produits correspondants sont additionnés pour obtenir l'élément de sortie. Le filtre continue de parcourir l'entrée, d'effectuer des convolutions et d'obtenir les éléments de sortie [19].

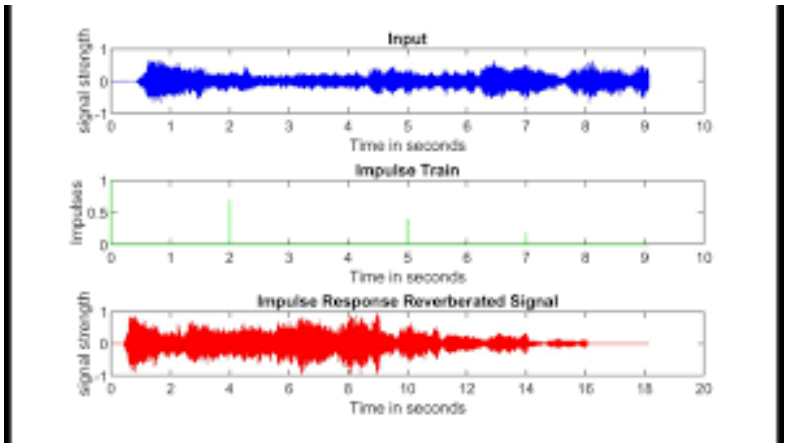


Figure 18 : Générer un écho dans l'audio en utilisant la convolution

3.2 Cartes de caractéristiques et noyaux de convolution

La convolution est courante en traitement de son. Elle consiste en une opération de multiplication de deux matrices de tailles différentes (généralement une petite et une grande), mais de même format (p.ex. wav) semblable, produisant une nouvelle matrice (également de même format). La convolution est donc le traitement d'une matrice par une autre petite matrice appelée matrice de convolution ou noyau (kernel) [18]. Le filtre parcourt toute la matrice principale de manière incrémentale et génère une nouvelle matrice constituée des résultats de la multiplication. Notez qu'il y a une marge dans la matrice finale pour laquelle nous ne pouvons pas calculer de valeur. Dans le traitement de son, ceci est utilisé par exemple pour effectuer un flou gaussien, ou détourer les éléments d'un audio.

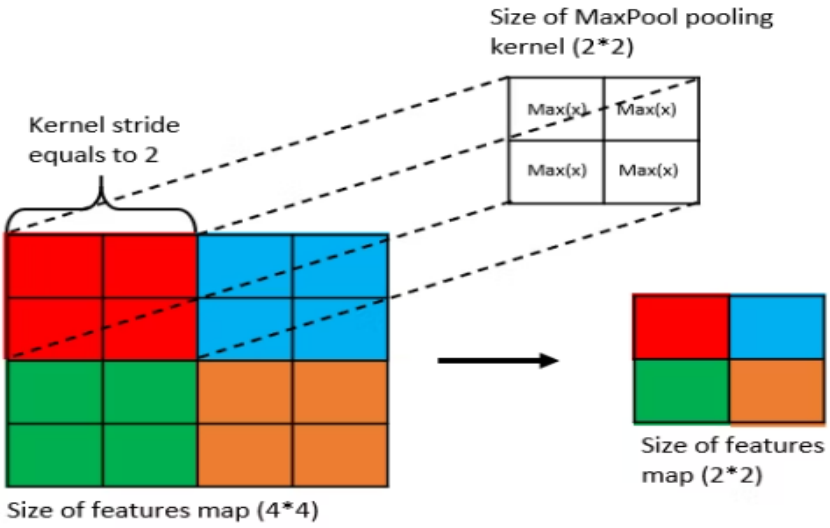


Figure 19 : Noyaux de convolution sur les réseaux de neurones convolutifs.

3.3 Architecture de CNN

L'architecture du Convolutional Neural Network dispose en amont d'une partie convolutive et comporte par conséquent deux parties bien distinctes [20] :

- **Une partie convolutive** : Son objectif final est d'extraire des caractéristiques propres à chaque audio en les compressant de façon à réduire leur taille initiale. En résumé, le son fourni en entrée passe à travers une succession de filtres, créant par la même occasion de nouveaux sons appelés cartes de convolutions. Enfin, les cartes de convolutions obtenues sont concaténées dans un vecteur de caractéristiques appelé code CNN.
- **Une partie classification** : Le code CNN obtenu en sortie de la partie convolutive est fourni en entrée dans une deuxième partie, constituée de couches entièrement connectées appelées perceptron multicouche (MLP pour Multi Layer Perceptron). Le rôle de cette partie est de combiner les caractéristiques du code CNN afin de classer l'audio. Pour revenir sur cette partie

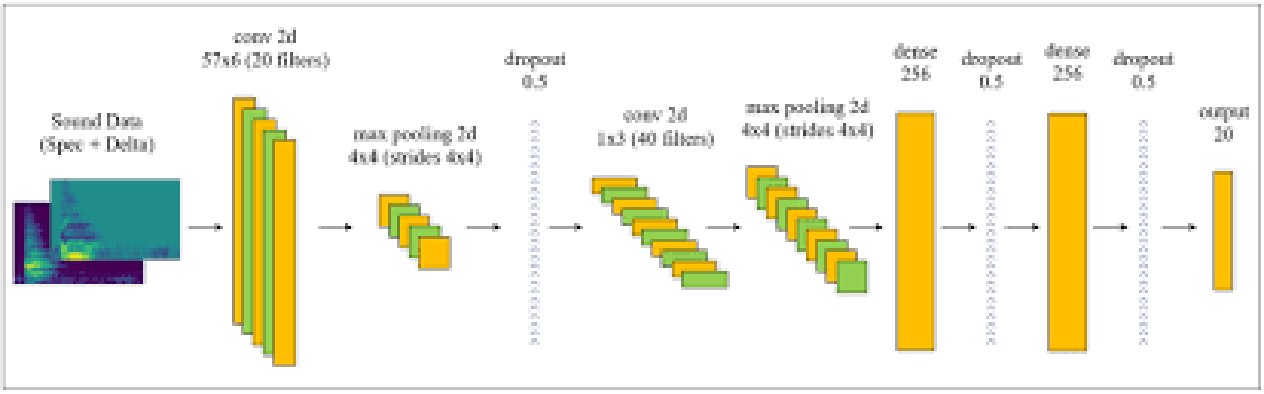


Figure20 : L'architecture de CNN utilisée pour la classification sonore.

Nous allons apprendre à classer les sons et à prédire la catégorie de ce son

Notre objectif principal est :

-D'apprendre à classer les sons et à prédire la catégorie de ce son. Ensuite d'acquérir une compréhension définitive du projet de classification audio tout en apprenant les concepts de base essentiels du traitement du signal et certaines des meilleures techniques utilisées pour obtenir les résultats souhaités.

-De donner un fichier audio en entrée, puis notre modèle doit déterminer si les caractéristiques audio contiennent l'une des étiquettes cibles.

Ces étiquettes de classe peuvent souvent être obtenues à partir d'une partie du nom de fichier de l'échantillon audio ou du nom du sous-dossier dans lequel se trouve le fichier. Alternativement, les étiquettes de classe sont spécifiées dans un fichier de métadonnées séparé, généralement au format TXT, JSON ou CSV

CHAPITRE 3

Implémentation du code

Chapitre 3

Implémentation du code

10. Architecture du CNN

Bien que les CNN soient principalement associés à des tâches de vision par ordinateur, ils peuvent également être adaptés à des tâches de traitement audio, telles que la reconnaissance vocale ou la classification musicale. La principale différence réside dans les données d'entrée et la manière dont elles sont traitées. Voici un aperçu d'une architecture CNN typique pour l'audio :

1.1. Représentation d'entrée

Les données audio sont généralement représentées sous la forme d'une forme d'onde, qui est un signal dans le domaine temporel. Pour convertir la forme d'onde dans un format adapté aux CNN, une approche courante consiste à utiliser des spectrogrammes. Les spectrogrammes représentent le contenu fréquentiel du signal audio dans le temps et sont obtenus en appliquant la transformée de Fourier à de petites fenêtres se chevauchant de la forme d'onde.

1.2. Couches convolutives

Semblables au traitement d'image, les couches convolutives dans un CNN audio consistent en des filtres apprenables qui glissent sur le spectrogramme d'entrée. Cependant, au lieu de convolutions bidimensionnelles, des convolutions unidimensionnelles sont utilisées pour tenir compte de la nature temporelle des données audio. Ces filtres capturent les modèles locaux et les dépendances dans les dimensions de fréquence et de temps du spectrogramme.

1.3. Fonction d'activation

Après chaque couche convolutive, une fonction d'activation, telle que ReLU ou sigmoïde, est appliquée élément par élément pour introduire la non-linéarité.

1.4. Regroupement de couches

Le regroupement de couches, généralement à l'aide d'un regroupement maximal, peut être incorporé pour réduire la dimension temporelle des cartes d'entités et capturer les caractéristiques les plus saillantes. Semblable au traitement d'image, la mise en commun maximale sélectionne la valeur maximale dans chaque région et supprime le reste.

1.5. Couches entièrement connectées

Après plusieurs couches de convolution et de regroupement, la sortie est aplatie en un vecteur unidimensionnel et passée à travers une ou plusieurs couches entièrement connectées. Ces couches capturent des représentations de haut niveau des données audio en apprenant des combinaisons complexes de fonctionnalités de bas niveau.

1.6. Couche de sortie

La dernière couche entièrement connectée est suivie d'une couche de sortie, qui utilise une fonction d'activation appropriée en fonction de la tâche. Par exemple, pour la reconnaissance vocale, une fonction d'activation softmax est couramment utilisée pour la classification multi-classes sur différents phonèmes ou mots.

1.7. Formation

Le processus de formation consiste à ajuster les poids et les biais du réseau à l'aide de la rétropropagation et de la descente de gradient stochastique. Le réseau apprend à minimiser la différence entre la sortie prédite et les véritables étiquettes à l'aide d'une fonction de perte appropriée, telle que l'entropie croisée.

Il convient de noter qu'il existe d'autres architectures spécialisées conçues spécifiquement pour le traitement audio, telles que WaveNet et Listen, Attend and Spell (LAS). Ces architectures exploitent des techniques telles que les convolutions dilatées, les réseaux de neurones récurrents (RNN) et les mécanismes d'attention pour capturer les dépendances à longue portée et gérer les entrées de longueur variable. Le choix de l'architecture dépend de la tâche audio spécifique et des ressources disponibles.

2. Prétraitement des données

2.1. Charger les données sur GoogleColab

Tout d'abord, Ce code est utilisé pour télécharger des fichiers de l'ordinateur vers Google Colab. La fonction « files.upload() » affiche une boîte de dialogue qui permet de sélectionner les fichiers à télécharger.

```
from google.colab import files
upload = files.upload()
import pandas as pd
data = pd.read_csv("UrbanSound8K.csv")
```

Comme il s'agit d'un fichier CSV, nous pouvons utiliser Pandas pour le lire. Ce fichier contient la metadata (contient des informations sur chaque échantillon audio du dataset).

-Afficher des informations pour metadata.

```
#from google.colab import drive
#drive.mount('/content/gdrive', force_remount=True)
import wave
obj = wave.open("audio_file.wav", "rb")
print("Number of channels", obj.getnchannels())
print("sample rate", obj.getframerate())
print("Number of frames", obj.getnframes())


t_audio = obj.getnframes() / obj.getframerate()
print(t_audio)
```

-Crée une nouvelle colonne appelée ‘chemin_relatif’ qui donne le chemin relatif vers le fichier à partir du répertoire racine de l'ensemble de données.

```
data['chamin_relatif'] = '/fold' + data['fold'].astype(str) + '/' + data['s
data.head()
```

data['slice_file_name'].astype(str) : convertit la colonne « slice_file_name » en type string.

-Après je sélectionne uniquement les colonnes « chemin_relatif» et "classID" et affectera la trame de données résultante aux données.

```
data = data[['chemin_relatif', 'classID']]
data.head()
```

-J'ai défini une classe « AudioUtil »qui contient une méthode statique « open() » pour charger un fichier audio et renvoyer le signal sous forme de tenseur et la fréquence d'échantillonnage.

```
import torchaudio


class AudioUtil:
    @staticmethod
    def open(file_path):
        waveform, sample_rate = torchaudio.load(file_path)
        return waveform, sample_rate
```

La méthode « open() » prend un paramètre file_path, qui est le chemin vers le fichier audio que je veux charger. Il utilise la fonction « torchaudio.load() » pour charger le fichier audio. Cette fonction renvoie un tuple contenant la forme d'onde (signal) et la fréquence d'échantillonnage. La forme d'onde est stockée dans la variable waveform et la fréquence d'échantillonnage est stockée dans la variable sample_rate.

-Cela chargera le fichier audio situé dans"/content/drive…" et attribuera la forme d'onde et la fréquence d'échantillonnage aux variables respectives.

```
audio_util = AudioUtil()
waveform, sample_rate = audio_util.open("/content/drive/MyDrive/cc/archive
```

2.2. Convertir en deux canaux

-La méthode « rechannel » prend en entrée une paire de tenseurs « PyTorch » qui représentent un signal audio et son taux d'échantillonnage. Le premier tenseur « sig » a une taille de (c, n) où c représente le nombre de canaux (1 pour mono, 2 pour stéréo) et n représente le nombre d'échantillons. Le second tenseur sr représente le taux d'échantillonnage du signal audio.

```
@staticmethod
def rechannel(audio, new_channel):
  sig, sr = audio
```

-La méthode prend également en entrée un nouveau nombre de canaux « new_channel ». Si le signal audio a déjà le nombre de canaux requis, la méthode ne fait rien.

```
if (sig.shape[0] == new_channel):
    # rien à faire

 return audio
```

2.2.1 Conversion de stéréo en mono

Supposons que nous ayons un signal audio stéréo avec deux canaux représentés par le tenseur « sig » de taille (2, n) où n est le nombre d'échantillons. Nous pouvons convertir ce signal en mono en sélectionnant simplement le premier canal, comme illustré ci-dessous :

```
sig = torch.tensor([[0.1, 0.2, 0.3, 0.4], [0.5, 0.6, 0.7, 0.8]])
sr = 44100
aud = (sig, sr)
aud_mono = AudioUtil.rechannel(aud, 1)
```

Le tenseur « aud_mono » renvoyé aura une taille de (1, n) et ne contiendra que les échantillons du premier canal :

```
>>> aud_mono
(tensor([[0.1000, 0.2000, 0.3000, 0.4000]]), 44100)
```

2.2.2. Conversion de mono en stéréo

Supposons maintenant que nous ayons un signal audio mono avec un seul canal représenté par le tenseur sig de taille (1, n) où n est le nombre d'échantillons. Nous pouvons convertir ce signal en stéréo en dupliquant simplement le canal existant, comme illustré ci-dessous :

```
sig = torch.tensor([[0.1, 0.2, 0.3, 0.4]])
sr = 44100
aud = (sig, sr)
aud_stereo = AudioUtil.rechannel(aud, 2)
```

Le tenseur « aud_stereo » renvoyé aura une taille de (2, n) et contiendra deux canaux identiques :

```
>>> aud_stereo
(tensor([[0.1000, 0.2000, 0.3000, 0.4000],
         [0.1000, 0.2000, 0.3000, 0.4000]]), 44100)
```

2.3. Standardiser le taux d'échantillonnage

-Cette ligne définit une méthode statique appelée resample qui prend deux arguments, aud et newsr.

```
@staticmethod
def resample(aud, newsr):
```

-Cette ligne décompresse le tuple d'argument ‘aud’en deux variables, sig et sr. sig est un tenseur 2D représentant un signal audio avec forme (num_channels, num_samples), où num_channels est le nombre de canaux audio (1 pour mono, 2 pour stéréo, etc.) et num_samples est le nombre d'échantillons audio dans chaque canal. sr est un entier représentant la fréquence d'échantillonnage du signal audio, en échantillons par seconde.

```
sig, sr = aud
```

-Ce bloc de code vérifie si la fréquence d'échantillonnage d'origine sr est égale à la fréquence d'échantillonnage souhaitée newsr. S'ils sont identiques, cela signifie que le signal audio est déjà à la fréquence d'échantillonnage souhaitée, de sorte que la méthode renvoie simplement le tuple audio d'origine tel quel.

```
if (sr == newsr):
    # Nothing to do
    return aud
```

-Cette ligne extrait le nombre de canaux audio du tenseur de signal.

```
num_channels = sig.shape[0]
```

-Cette ligne utilise la transformation « Resample » de la bibliothèque « torchaudio » pour ré échantillonner le premier canal du signal audio « sig » de la fréquence d'échantillonnage d'origine à la nouvelle fréquence d'échantillonnage souhaitée. Le résultat est un nouveau tenseur « resig » avec le même nombre d'échantillons que le canal d'origine, mais avec un nouveau taux d'échantillonnage. La syntaxe d'indexation « sig[:1,:] » sélectionne le premier canal du tenseur sig.

```
# Resample first channel
resig = torchaudio.transforms.Resample(sr, newsr)(sig[:1,:])
```

-Ce bloc de code vérifie si le signal audio a plus d'un canal. Si c'est le cas, il utilise la transformation « Resample » pour ré échantillonner le deuxième canal du signal audio de la même manière que le premier canal, puis concatène les deux canaux ré échantillonnés en un seul tenseur à l'aide de la fonction torch.cat. La syntaxe d'indexation « sig[1:,:] » sélectionne tous les canaux du tenseur sig à l'exception du premier.

```
if (num_channels > 1):
    # Resample the second channel and merge both channels
    retwo = torchaudio.transforms.Resample(sr, newsr)(sig[1:,:])
    resig = torch.cat([resig, retwo])
```

-Cette ligne renvoie un tuple contenant le « resig » du tenseur du signal audio ré échantillonné et le « newsr » du taux d'échantillonnage souhaité. Le tenseur de signal audio ré échantillonné a la même forme que le tenseur de signal d'origine, mais avec une nouvelle fréquence d'échantillonnage.

```
return ((resig, newsr))
```

2.4. Redimensionner à la même longueur

-La méthode « pad_trunc() » prend deux paramètres : audio, qui est un tuple contenant le signal et la fréquence d'échantillonnage, et « max_ms », qui est la longueur maximale en millisecondes à laquelle le signal doit être rempli ou tronqué.

La variable « max_len » est calculée en multipliant la fréquence d'échantillonnage par « max_ms » et en divisant par 1000 pour convertir les millisecondes en échantillons.

```
@staticmethod
def pad_trunc(audio, max_ms):
    sig, sr = audio
    num_rows, sig_len = sig.shape
    max_len = sr//1000 * max_ms
```

-Si la longueur du signal « sig_len » est supérieure à « max_len », le signal esttronqué en conservant les premiers « max_len » échantillons.

```
    if (sig_len > max_len):
        sig = sig[:,:max_len]
```

-Si la longueur du signal est inférieure à « max_len », un remplissage est ajouté au début et à la fin du signal. Les longueurs du rembourrage sont déterminées de manière aléatoire dans la plage autorisée.

```
elif (sig_len < max_len):
    pad_begin_len = random.randint(0, max_len - sig_len)
    pad_end_len = max_len - sig_len - pad_begin_len
    pad_begin = torch.zeros((num_rows, pad_begin_len))
    pad_end = torch.zeros((num_rows, pad_end_len))
    sig = torch.cat((pad_begin, sig, pad_end), 1)
```

Zéro tenseurs de dimensions appropriées sont créés pour le rembourrage, c'est-à-dire « pad_begin » et « pad_end ».Le signal est concaténé avec les tenseurs de remplissage à l'aide de « torch.cat() » le long de la deuxième dimension (axe du temps) pour s'assurer qu'ils ont la même longueur.

-Je peux appeler la méthode « pad_trunc() » sur un signal audio, en fournissant la longueur maximale en millisecondes comme argument. Voici un exemple

```
audio = (sig, sr)  # Provide the audio signal and sample rate
max_ms = 5000  # Maximum length in milliseconds
padded_truncated_audio = AudioUtil.pad_trunc(audio, max_ms)
```

La variable « padded_truncated_audio » contiendra le signal audio modifié et la fréquence d'échantillonnage après l'application du rembourrage ou de la Troncature.

3. Augmentation des données : décalage temporel

La méthode prend deux arguments : audio et « shift_limit » qui spécifie la limite maximale pour le décalage temporel.

```
@staticmethod
def time_shift(audio, shift_limit):
    sig,sr = audio
    _, sig_len = sig.shape
    shift_amt = int(random.random() * shift_limit * sig_len)
    return (sig.roll(shift_amt), sr)
```

Ensuite, une quantité de décalage aléatoire est calculée en utilisant « random.random() * shift_limit * sig_len ». « random.random() » génère une valeur aléatoire à virgule flottante, et la multiplier par « shift_limit * sig_len» met à l'échelle la valeur aléatoire en fonction des limites spécifiées et de la longueur du signal. Il effectue l'opération de décalage temporel sur le tenseur de signal en utilisant la méthode du roulement(roll). Cette méthode décale les éléments du tenseur de la quantité spécifiée le long de la dimension spécifiée. Cela crée l'effet de décalage de l'audio dans le temps.

3.1. Spectrogramme Mel

-Définir une méthode qui prend un signal audio, calcule le spectrogramme Mel et le convertit en décibels à l'aide du module "transforms" de la bibliothèque «librosa». Le spectrogramme résultant est ensuite renvoyé.

Ce code définit une méthode statique nommée «spectro_gram» qui prend une entrée audio ainsi que des paramètres optionnels. Il affecte les valeurs audio aux variables sig et sr et définit «top_db» sur 80, ce qui représente le seuil des valeurs en décibels.

```
@staticmethod
def spectro_gram(audio, n_mels=64, n_fft=1024, hop_len=None):
    sig, sr = audio
    top_db = 80
```

-Crée un spectrogramme Mel en utilisant la classe. Il prend le signal audio sig en entrée et définit des paramètres tels que «sr», le nombre de points FFT «n_fft», la longueur de saut

«hop_len» et le nombre de cases de fréquence Mel «n_mels». L'objet «MelSpectrogram» est alors appelé avec pour calculer le spectrogramme, et le résultat est assigné à la variable «spec».

```
spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n
```

-Cette ligne applique une transformation pour convertir le spectrogramme d'amplitude en décibels à l'aide de la classe «transforms.AmplitudeToDB». Il définit le paramètre top_db sur 80. L'objet «AmplitudeToDB» est ensuite appelé avec spec comme argument pour effectuer la conversion, et le résultat est réattribué à la variable spec.

```
    spec = transforms.AmplitudeToDB(top_db=top_db)(spec)
```

3.2. Augmentation des données : masquage temporel et fréquentiel

Ce code définit une méthode statique nommée «spectro_augment» qui prend une spécification de spectrogramme en entrée, ainsi que les paramètres facultatifs «max_mask_pct», «n_freq_masks» et «n_time_masks». La forme du spectrogramme spec est décompressée pour obtenir le nombre de cases de fréquence Mel (n_mels) et le nombre de pas de temps «n_steps». La valeur moyenne du spectrogramme spec est calculée et affectée à la variable «mask_value». La variable «aug_spec» est initialisée avec la valeur de spec.

```
@staticmethod
def spectro_augment(spec, max_mask_pct=0.1, n_freq_masks=1, n_time_masks=1)
    _, n_mels, n_steps = spec.shape
    mask_value = spec.mean()
    aug_spec = spec
```

-Cette section applique l'augmentation de masquage de fréquence au spectrogramme. Il calcule le paramètre de masque de fréquence en pourcentage du nombre de tranches de fréquence Mel. La classe «transforms.FrequencyMasking» est alors appelée dans une boucle «n_freq_masks» fois. Il applique la transformation de masquage de fréquence au spectrogramme «aug_spec», en utilisant «freq_mask_param» et«mask_value» comme arguments. Le spectrogramme augmenté résultant est réaffecté à « aug_spec ».

```
    freq_mask_param = max_mask_pct * n_mels
    for _ in range(n_freq_masks):
        aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, m
```

-De même, cette section applique l'augmentation de masquage temporel au spectrogramme. Il calcule le paramètre de masque temporel en pourcentage du nombre de pas de temps.

```
time_mask_param = max_mask_pct * n_steps
for _ in range(n_time_masks):
    aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_va
```

Généralement, ce code effectue une augmentation des données sur un spectrogramme donné en appliquant des transformations de masquage de fréquence et de masquage temporel à l'aide du module de transformations de la bibliothèque librosa. Les paramètres contrôlent l'étendue du masquage et le spectrogramme augmenté est renvoyé en sortie.

4. Définir un chargeur de données personnalisé

-Ce sont les instructions d'importation nécessaires pour utiliser les bibliothèques requises.

```
from torch.utils.data import DataLoader, Dataset, random_split
import torchaudio
```

-Ce code définit une classe de jeu de données personnalisée nommée «SoundDS» qui hérite de la classe Dataset fournie par «PyTorch». Il a une méthode d'initialisation «__init__» qui prend une trame de données «df» et un chemin de données «data_path» comme entrées. Il initialise également certains paramètres tels que «duration» (la durée souhaitée de l'audio), «sr» (la fréquence d'échantillonnage souhaitée), «channel» (le nombre souhaité de canaux audio) et «shift_pct» (le pourcentage de décalage temporel pour l'augmentation des données).

-Cette méthode renvoie la longueur de l'ensemble de données, qui correspond au nombre d'éléments dans la trame de données « self.df ».

```
def __len__(self):
  return len(self.df)
```

Cette méthode renvoie l'élément à l'index donné «idx» dans l'ensemble de données. Il construit d'abord le chemin de fichier absolu du fichier audio basé sur le «data_path» et le chemin relatif stocké dans le «dataframe». Il charge ensuite l'audio à l'aide de la fonction «AudioUtil.open». Diverses fonctions de traitement audio de la classe AudioUtil sont appliquées à l'audio, y compris le ré échantillonnage, la recanalisation, le rembourrage/troncature, le décalage temporel et la génération du spectrogramme. L'augmentation des données est également effectuée en appliquant un masquage de spectrogramme. Le spectrogramme augmenté et l'ID de classe correspondant sont renvoyés sous forme de tuple.

```
def __getitem__(self, idx):
  audio_file = self.data_path + self.df.loc[idx, 'relative_path']
  class_id = self.df.loc[idx, 'classID']

  aud = AudioUtil.open(audio_file)
  reaud = AudioUtil.resample(aud, self.sr)
  rechan = AudioUtil.rechannel(reaud, self.channel)

  dur_aud = AudioUtil.pad_trunc(rechan, self.duration)
  shift_aud = AudioUtil.time_shift(dur_aud, self.shift_pct)
  sgram = AudioUtil.spectro_gram(shift_aud, n_mels=64, n_fft=1024, hop_le
  aug_sgram = AudioUtil.spectro_augment(sgram, max_mask_pct=0.1, n_freq_m

  return aug_sgram, class_id
```

-Cette ligne crée une instance de la classe «SoundDS» en transmettant le dataframe «df» et le chemin de données «data_path».

```
myds = SoundDS(df, data_path)
```

Ces lignes calculent le nombre d'éléments dans l'ensemble de données et déterminent le nombre d'éléments pour la formation «num_train» et la validation «num_val». La fonction «random_split» est ensuite utilisée pour diviser l'ensemble de données de manière aléatoire en ensembles de données d'apprentissage et de validation, avec les proportions spécifiées.

```
num_items = len(myds)
num_train = round(num_items * 0.8)
num_val = num_items - num_train
train_ds, val_ds = random_split(myds, [num_train, num_val])
```

-Ce sont les instructions d'importation nécessaires pour utiliser les modules

et fonctions PyTorch requis.

```
import torch.nn.functional as F
from torch.nn import init
```

- Définissez la classe AudioClassifier, qui est une sous-classe de nn.Module

```
class AudioClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        conv_layers = []
```

-Initialisez une liste vide «conv_layers» pour stocker les couches convolutionnelles.

-Définissez le premier bloc convolutif avec activation «ReLU» et normalisation par lots :

```
self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2
self.relu1 = nn.ReLU()
self.bn1 = nn.BatchNorm2d(8)
init.kaiming_normal_(self.conv1.weight, a=0.1)
self.conv1.bias.data.zero_()
conv_layers += [self.conv1, self.relu1, self.bn1]
```

Ce bloc prend une entrée avec 2 canaux et produit 8 canaux de sortie. La fonction «kaiming_normal _» initialise les poids à l'aide de la méthode d'initialisation normale de Kaiming, et le biais est initialisé à zéro.

-Définissez trois autres blocs convolutionnels (conv2, conv3, conv4) similaires au premier bloc, mais avec des tailles de canaux d'entrée et de sortie différentes.

-Définissez le classificateur linéaire avec une couche de regroupement moyenne adaptative et une couche linéaire entièrement connectée

```
self.ap = nn.AdaptiveAvgPool2d(output_size=1)
self.lin = nn.Linear(in_features=64, out_features=10)
```

La couche de regroupement moyen adaptatif réduit les dimensions spatiales du tenseur d'entrée à une taille fixe, et la couche linéaire effectue la classification finale avec 64 caractéristiques d'entrée et 10 classes de sortie.

-Enveloppez les blocs convolutifs dans un module séquentiel

```
self.conv = nn.Sequential(*conv_layers)
```

Cela permet l'exécution séquentielle des couches convolutionnelles.

-Définissez la méthode directe pour définir les calculs de passe directe du modèle

```
def forward(self, x):
    x = self.conv(x)
    x = self.ap(x)
    x = x.view(x.shape[0], -1)
    x = self.lin(x)
    return x
```

L'entrée x est passée à travers les blocs convolutifs, suivis de la couche de mise en commun moyenne adaptative. Le tenseur résultant est aplati puis passé à travers la couche linéaire pour obtenir la sortie finale.

-Créez une instance du modèle AudioClassifier

```
myModel = AudioClassifier()
```

-Vérifiez si le GPU est disponible et déplacez le modèle vers le GPU s'il l'est

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
myModel = myModel.to(device)
```

-Vérifiez que le modèle se trouve sur le GPU

```
next(myModel.parameters()).device
```

Cela imprime l'appareil où se trouvent les paramètres du modèle.

C'est l'explication étape par étape du code. Le modèle «AudioClassifier» se compose de blocs convolutifs suivis d'une couche de regroupement moyenne adaptative et d'un classificateur linéaire, ce qui le rend adapté aux tâches de classification audio.

5. Apprentissage (entrainement)

-La fonction « training »prend trois arguments : « model » (le modèle à former), « train_dl » (le chargeur de données de formation) et «num_epochs» (le nombre d'époques de formation). Il initialise la fonction de perte à l'aide de « nn.CrossEntropyLoss() », l'optimiseur à l'aide de

« torch.optim.Adam() » avec un taux d'apprentissage de 0,001 et le planificateur de taux d'apprentissage à l'aide de « torch.optim.lr_scheduler.OneCycleLR() ». La fonction entre dans une boucle pour chaque époque, où elle initialise les variables pour suivre la perte en cours, corriger les prédictions et les prédictions totales.

```
def training(model, train_dl, num_epochs):
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001,
                                                  steps_per_epoch=int(len(tra
                                                  epochs=num_epochs,
                                                  anneal_strategy='linear')
```

-À l'intérieur de la boucle d'époque, il itère sur les mini-lots des données d'apprentissage à l'aide de « enumerate(train_dl) ».

-Pour chaque mini-lot, il récupère les entités d'entrée (inputs) et les étiquettes cibles « labels », et les transfère vers l'appareil «device» approprié.

```
  for i, data in enumerate(train_dl):
      inputs, labels = data[0].to(device), data[1].to(device)
```

-Les entités en entrée sont normalisées en calculant leur moyenne «inputs_m» et leur écart type « inputs_s », puis en soustrayant la moyenne et en divisant par l'écart type.

```
inputs_m, inputs_s = inputs.mean(), inputs.std()
inputs = (inputs - inputs_m) / inputs_s
```

-Les gradients des paramètres du modèle sont mis à zéro à l'aide « optimizer.zero_grad () » pour éviter l'accumulation des précédents itérations.

```
    optimizer.zero_grad()
```

-Le modèle est appelé avec les entrées normalisées pour obtenir les prédictions de sortie « outputs ».

-La perte entre les prédictions et les étiquettes cibles est calculée à l'aide de la fonction de perte d'entropie croisée « criterion ».

-La rétro propagation est effectuée en appelant «loss.backward()»pour calculer les gradients pour tous les paramètres du modèle.

-L'optimiseur effectue une étape pour mettre à jour les paramètres du modèle en fonction des gradients calculés à l'aide de l‘ «optimiser.step()».

-Le planificateur de taux d'apprentissage est également mis à jour à l'aide de «scheduler.step()» pour ajuster le taux d'apprentissage en fonction de la progression de la formation.

```
outputs = model(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
scheduler.step()
```

-La perte courante est mise à jour en ajoutant la perte actuelle du mini-lot à «running_loss».

```
running_loss += loss.item()
```

-Les classes prédites sont obtenues en trouvant les indices des valeurs maximales dans les prédictions de sortie à l'aide de « torch.max(outputs, 1) ». Le nombre de prédictions correctes est incrémenté en additionnant le nombre de prédictions correspondantes entre les classes prédites et les étiquettes cibles. Le nombre total de prédictions est incrémenté de la taille du mini-lot « prediction.shape[0] ».

```
_, prediction = torch.max(outputs, 1)
correct_prediction += (prediction == labels).sum().item()
total_prediction += prediction.shape[0]
```

Après avoir traité tous les mini-lots, la perte moyenne et la précision pour l'époque sont calculées.

-Le nombre d'époques, la perte moyenne et la précision sont imprimés à l'aide de «print(f'Epoch : {epoch}, Loss : {avg_loss:.2f}, Accuracy : {acc:.2f}‘».

```
num_batches = len(train_dl)
avg_loss = running_loss / num_batches
acc = correct_prediction / total_prediction
print(f'Epoch: {epoch}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')
```

-La boucle d'apprentissage se poursuit pendant le nombre d'époques spécifié.

-Une fois que toutes les époques sont terminées, le message « Finished Training » est imprimé.

Enfin, la variable « num_epochs » est définie sur 2 et la fonction d'entraînement est appelée avec le modèle fourni « myModel », le chargeur de données d'entraînement « train_dl » et le nombre d'époques.

```
    print('Finished Training')

num_epochs = 2
training(myModel, train_dl, num_epochs)
```

6. Inférence

-La fonction « inference » prend deux arguments : « model » (le modèle formé) et « val_dl » (le chargeur de données de validation).

-Il initialise les variables pour suivre le nombre de prédictions correctes « correct_prediction » et le nombre total de prédictions « total_prediction ».

```
def inference(model, val_dl):
  correct_prediction = 0
  total_prediction = 0
```

-Le gestionnaire de contexte torch.no_grad() est utilisé pour désactiver les mises à jour de gradient et économiser de la mémoire pendant l'inférence. Cela signifie que les gradients ne seront pas calculés pour la passe avant.

-La fonction itère sur les mini-lots des données de validation à l'aide de «val_dl».

-Pour chaque mini-lot, il récupère les entités d'entrée « inputs » et les étiquettes cibles «labels», et les transfère vers l'appareil « device » approprié.

```
with torch.no_grad():
  for data in val_dl:
    inputs, labels = data[0].to(device), data[1].to(device)
```

-Les caractéristiques d'entrée sont normalisées de la même manière que dans le code d'entraînement en calculant la moyenne « inputs_m » et l'écart type « inputs_s » et en normalisant les entrées.

```
  inputs_m, inputs_s = inputs.mean(), inputs.std()
  inputs = (inputs - inputs_m) / inputs_s
```

-Le modèle est appelé avec les entrées normalisées pour obtenir les prédictions de sortie « outputs ».

```
  outputs = model(inputs)
```

-Les classes prédites sont obtenues en trouvant les indices des valeurs maximales dans les prédictions de sortie à l'aide de « torch.max(outputs, 1) ».

-Le nombre de prédictions correctes est incrémenté en additionnant le nombre de prédictions correspondantes entre les classes prédites et les étiquettes cibles.

-Le nombre total de prédictions est incrémenté de la taille du mini-lot « prediction.shape[0] ».

```
_, prediction = torch.max(outputs, 1)
correct_prediction += (prediction == labels).sum().item()
total_prediction += prediction.shape[0]
```

-Après traitement de tous les mini-lots, la précision est calculée en divisant le nombre de prédictions correctes par le nombre total de prédictions.

-La précision et le nombre total d'éléments dans l'ensemble de données de validation sont imprimés à l'aide de « print(f'Accuracy : {acc:.2f}, Total items :{total_prediction}') ».

```
acc = correct_prediction / total_prediction
print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')
```

Enfin, la fonction d'inférence est appelée avec le modèle formé « myModel » et le chargeur de données de validation « val_dl ». La précision du modèle sur le jeu de données de validation est calculée et imprimée.

```
inference(myModel, val_dl)
```

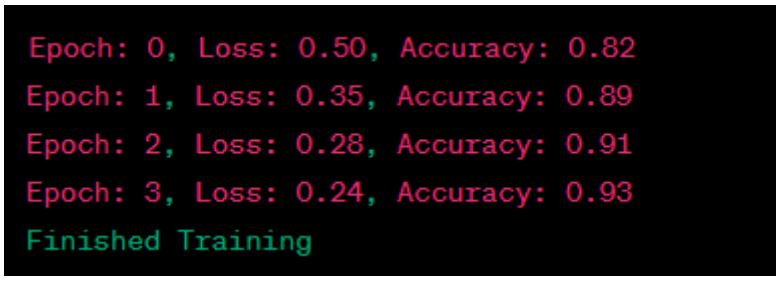
7. Résultats

Pour **l'apprentissage (Entraînement)**, nous avons formés le modèle pour plusieurs époques.
Pour tester on a pris époques = 2 et époques = 4.
En exécutant, notre code affichera les statistiques de perte (loss) et de précision (accuracy) à la fin de chaque époque d'entraînement.

La sortie atteinte : Cas époques = 2 (num_epochs=2) :

```
Epoch: 0, Loss: 1.23, Accuracy: 0.65
Epoch: 1, Loss: 0.95, Accuracy: 0.72
Finished Training
```

La sortie atteinte : Cas époques = 4 (num_epochs=4) :



Ces statistiques indiquent la perte moyenne (loss) et la précision (accuracy) calculées sur l'ensemble de données d'entraînement à la fin de chaque époque. La précision est calculée en comparant les prédictions du modèle avec les étiquettes cibles réelles.

Pour **l'inférence**, Lorsque on appel la fonction 'inference', elle affiche deux informations :

1- L'exactitude (accuracy) : il s'agit du pourcentage de prédictions correctes parmi toutes les prédictions effectuées.
2- Le nombre total d'articles : il s'agit du nombre total d'articles dans l'ensemble de validation sur lesquels les prédictions ont été effectuées.

Cela signifie qu'il y a une exactitude de 85%, ce qui indique que 85% des prédictions effectuées sur l'ensemble de validation étaient correctes. De plus, le nombre total d'articles dans l'ensemble de validation est de 1000.

Conclusion Générale

Chaque jour, notre environnement urbain est rempli de sons variés tels que les bruits de climatiseurs, les klaxons de voitures, les jeux d'enfants, les aboiements de chiens, les bruits de forage, les moteurs au ralenti, les coups de feu, les marteaux-piqueurs, les sirènes et la musique de rue. Ces sons font partie intégrante de notre vie quotidienne. Alors que les humains sont capables de différencier facilement ces sons, les ordinateurs sont désormais capables de les classifier dans des catégories spécifiques grâce à l'apprentissage profond.

Dans ce projet de fin d'études, nous avons mis en œuvre une méthode basée sur l'apprentissage profond, plus précisément les réseaux de neurones convolutif, afin de permettre une classification automatique des sons urbains en catégories. Nous avons créé un modèle capable d'extraire des caractéristiques à partir des enregistrements audio des sons urbains contenant des exemples de différents types de bruits urbains préalablement étiquetés. En s'appuyant sur cette base d'apprentissage, le modèle est capable de généraliser et de classer automatiquement de nouveaux enregistrements audio dans les catégories appropriées, avec une précision calculée en comparant les prédictions du modèle avec les étiquettes cibles réelles. La précision est une mesure importante pour évaluer les performances d'un modèle de réseau neuronal. Elle indique la proportion de prédictions correctes par rapport au nombre total de prédictions. Au fur et à mesure de l'entraînement, la précision peut augmenter ou se stabiliser. Donc la précision est un indicateur clé, mais il est important de l'interpréter en conjonction avec d'autres métriques pour avoir une vue complète des performances du modèle.

Bibliographies:

Livre, monographie

[1] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143

[2] C.M. Bishop, “Neural Networks for Pattern Recognition”, Oxford University Press, 1995.

Articles de revue

[3] Jiaxing. Y., Takumi. K., Masahiro. M., Urban sound event classification based on local and global features aggregation, Applied Acoustics, Volume 117, Part B, 2017, Pages 246-256

[4] Nogueira, A.F.R.; Oliveira, H.S.; Machado, J.J.M.; Tavares, J.M.R.S. Sound Classification and Processing of Urban Environments: A Systematic Literature Review. *Sensors* 2022, *22*, 8608.https://doi.org/10.3390/s22228608

[5] G. Saint-Cirgue, "Apprendre le Machine Learning en une semaine "Tous droits réservés © 2019 Guillaume Saint-Cirgue machinelearnia.com 1,” 2019.

[6] G. Keifer and F. Effenberger, Big Data Et Machine Learning, vol. 6, no. 11. 1967.

[7] A. Chaudhary, K. S. Chouhan, J. Gajrani, and B. Sharma, Deep Learning WithPyTorch. 2020.

[8] J. J. Heckman, R. Pinto, and P. A. Savelyev, “Artificial Intelligence, IOT and Machine Learning” Angew. Chemie Int. Ed. 6(11), 951–952., 1967.

[9] C-A. Azencott, Introduction au Machine Learning, Dunod, France. 2018.

[10] H. Kinsley and D. Kukieła, “Neural Networks from Scratch in Python,” p. 658, 2020.

[11] R. Rakotomalala, “Ricco Rakotomalala Tutoriels Tanagra-http://tutoriels-datamining.blogspot.fr/ Perceptrons simples et multicouches,” [Online]. Available: http://tutoriels-data-mining.blogspot.fr/.2013

[12] A. Romero and A. Romero, “Assisting the training of deep neural networks with applications to computer vision assisting the training of deep neural networks with applications to computer vision.”University of Barcelona. 2015

[13] T. Okatani, Python Deep Learning 2nd, vol. 33, no. 2. 2015. Packt

[14] S. Kojouharov, “Cheat Sheets for AI, Neural Networks, Machine Learning, Deep Learning and Big Data,” Becom. Hum. Explor. AI, pp. 1–31, 2017, [Online]. Available: https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deeplearning-big-data-678c51b4b463.

[15] Y .LeCun “A theoretical framework for Back-Propagation ”G.Hinton, and T.Sejnowski, editors, Proceedings of the 1988 Connectionist Models Summer School p 21-28, CMU, Pittsburgh . 1988

Article d'actes de conférence

[16] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," in IEEE Signal Processing Letters, March 2017

Documents web

[17]https://towardsdatascience.com/urban-sound-classification-using-neural-networks-9b6fcd8a9150

[18] https://mikesmales.medium.com/sound-classification-using-deep-learning-8bc2aa1990b7

[19] https://librosa.org/doc/latest/generated/librosa.feature.melspectrogram.html

[20]https://www.analyticsvidhya.com/blog/2022/04/guide-to-audio-classification-using-deep-learning/