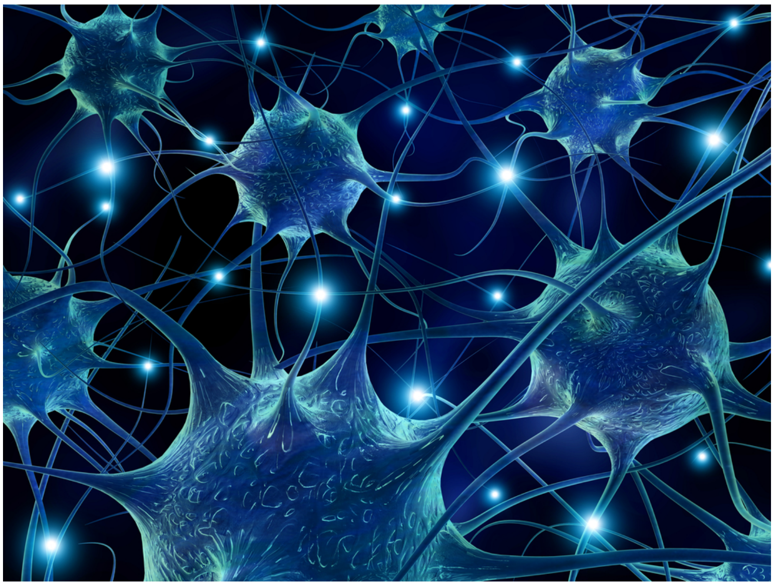
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
ROUEN NORMANDIE



Projet de Recherche Opérationnelle

Composition musicale par réseau de neurones

DRIGUEZ CLAIRE
CATELAIN Jeremy
RAMAGE Lucas
GM4

Tuteur : M. Knippel

Octobre - Décembre 2017

Table des matières

Introduction

Chapitre 1

Réseau de neurones et apprentissage

1.1 Réseau de neurones

1.1.1 Le neurone, un modèle spécifique

Un neurone est un mécanisme possédant une entrée, une unité de Processing et une sortie. C'est une fonction paramétrée non linéaire à valeurs bornées.

Les variables sur lesquelles opère le neurone sont appelées les entrées du neurone et la valeur de la fonction est désignée comme la sortie de la fonction. Ci-dessous est représenté un neurone représentant une fonction non linéaire paramétrée bornée $y = f(x, w)$ avec x les variables et w les paramètres.

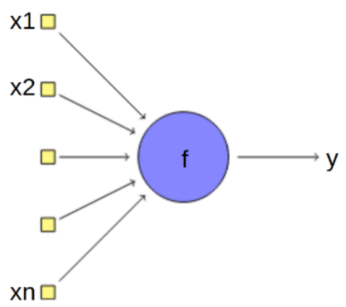


FIGURE 1.1 – Modélisation d'un neurone

L'entrée du neurone calcule la véritable variable d'entrée de l'unité de Processing en effectuant la somme des variables envoyées au mécanisme. Chaque variable envoyée au mécanisme est le produit entre une variable propre à un neurone précédent x_i et son paramètre w_i appelé le poids.

La valeur résultante peut alors être appelé le « potentiel » v tel que $v = \sum_{i \in I} w_i x_i + w_0$ avec w_0 appelé le biais ou seuil d'activation. Le seuil d'activation est propre à chaque neurone. Le biais w_0 peut être considéré comme un neurone avec comme variable $x_0 = 1$.

L'unité de Processing comporte une fonction d'activation f et un poids w_i'. Le processus consiste à appliquer cette fonction d'activation à la variable v et à considérer la valeur de sortie spécifique dépendant de la nature de f uniquement si le potentiel v est supérieur au seuil d'activation. Si c'est le cas, la valeur résultante est alors le produit entre le résultat de f appliquée à v et le poids w_i' propre au neurone i en question. Soit s la sortie tel que : $s = f(v) \cdot w_i'$.

La sortie consiste à considérer la valeur résultante s, si celle-ci est différente de zéro, comme une variable d'entrée pour le neurone suivant et à transmettre cette valeur à toutes les entrées des neurones suivants.

1.1.2 Les réseaux de neurones

On compte deux types de réseaux de neurones ; les réseaux à propagation avant ou réseaux de neurone acycliques et les réseaux de neurones cycliques. Un réseau de neurone est modélisé comme un graphe, adapté au problème en question. Les nœuds sont alors les neurones et les arêtes, les connexions entre ces neurones.

Le réseau à propagation avant réalise une ou plusieurs fonctions non linéaires de ses entrées par composition des fonctions réalisées par chacun de ses neurones. Les informations circulent des entrées vers les sorties sans retour en arrière. Les neurones effectuant le dernier calcul de la composition de fonctions sont appelés neurones de sorties et ceux effectuant des calculs intermédiaires sont appelés neurones cachés.

Perceptron multicouche

Le réseau à propagation avant le plus simplifié est le « Perceptrons multicouche » (Multi-Layer Perceptron). C'est un réseau de neurones dont les neurones cachés ont des fonctions d'activation sigmoïde.

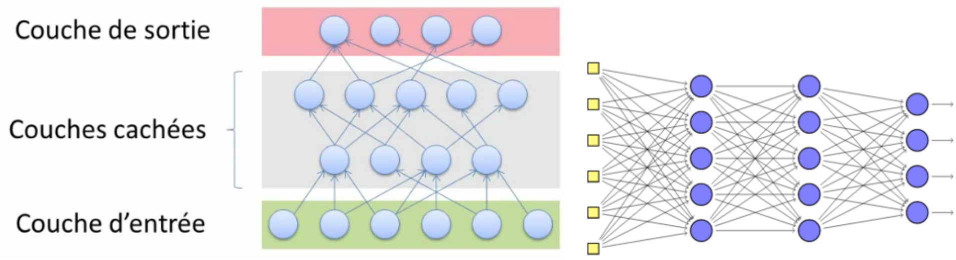


FIGURE 1.2 – Modèle de perceptron multicouche

La couche d'entrée représente les informations que l'on donne à l'entrée du réseau (exemple : pixel des images etc..). Les couches cachées permettent de donner une abstraction au modèle. Tous les arcs d'un nœud ont le même poids car chaque nœud a une valeur de sortie unique.

Il fait parti des algorithmes supervisés de classificateurs binaires. Celui-ci est constitué de neurones munie d'une « règle d'apprentissage » qui détermine les poids de manière automatique tel que $s = f(v).w_i'$ est la sortie. En fonction du résultat de s, on en déduit la réponse prédictive de l'objet en question.

Remarque : La notion de nœud est alors introduite. Les nœuds, qui peuvent appelés neurones, reçoivent une information de leurs prédécesseurs (les neurones de la couche précédente) et combinent cette information selon des pondérations identifiées par des poids w_i . Chaque nœud peut aussi posséder un seuil d'activation, appelé aussi biais, noté w_0. Le but est d'ajuster ces poids afin d'obtenir une sortie relativement correcte. Cela est réalisable lors de la phase d'apprentissage. Il faut aussi définir la qualité de chaque sortie donnée compte tenu de l'entrée. Cette valeur est appelé le coût (norme 2 par exemple avec la différence entre la réponse de la fonction et la sortie du réseau, au carré).

Une fois le coût calculé, la rétro-propagation peut être utilisée afin de réduire le calcul du gradient du coût par rapport au poids (c'est-à-dire la dérivée du coût par rapport à chaque poids pour chaque nœuds dans chaque couche). Ensuite, une méthode d'optimisation est utilisée pour ajuster les poids afin de réduire les coûts. Ces méthodes peuvent être retrouvées dans des bibliothèques et les gradients peuvent ainsi être alimentés par la bonne fonction et cette dernière, par la suite, ajuste les poids correctement.

La fonction sigmoïde

La fonction d'activation est définie comme suit :

$$\begin{cases} f(x_1,..x_p) = 1 & \text{si} \sum\limits_{i \in I} w_i x_i > w_0 \\ f(x_1,..,x_p) = 0 & \text{sinon} \end{cases}$$

avec b le seuil d'activation ou biais

Il s'agit de la fonction de Heaviside définie par $f(x_1,..,x_p) = H(\sum\limits_{i \in I} w_i x_i - w_0)$ mais celle-ci ne répond pas aux critères permettant d'utiliser la méthode du gradient car elle n'est pas dérivable et continue. De ce fait, la fonction d'activation généralement recommandée est la fonction sigmoïde (en forme de s) qui est symétrique par rapport à l'origine.

Elle est définie par :

$$f_1(x) = \frac{1}{1 + e^{-x}}$$

et plus généralement :

$$f_\lambda(x) = \frac{1}{1 + e^{-\lambda x}}$$

Remarque : si $\lambda = \frac{1}{T}$, si T tend vers 0, la fonction sigmoïde tend vers une fonction de Heaviside.

Voici l'allure de la courbe pour f_1 :

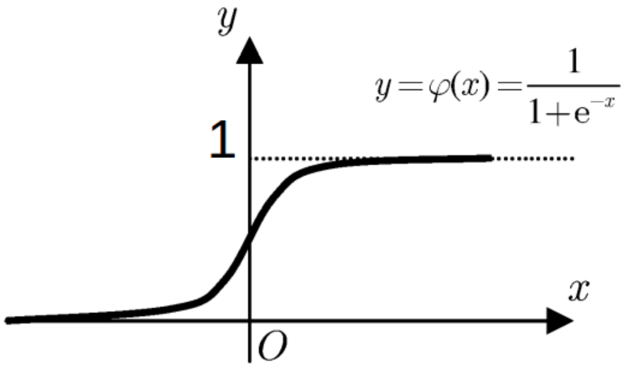


FIGURE 1.3 – Modèle de perceptron multicouche

Celle-ci possède des propriétés intéressantes. Celle-ci est continue et dérivable à l'infini. En effet, $f'_\lambda(x) = f(x) \cdot (1 - f(x))$ et $f \in C^\infty$. Le calcul de la dérivée de cette fonction en un point est directement calculable à partir de ce point, ce qui rend facilement applicable la méthode du gradient. De plus, la fonction renvoie des valeurs entre 0 et 1 donc l'interprétation en tant que probabilité est alors possible.

La fonction ReLu (Unité de Rectification Linéaire () peut aussi être utilisée comme fonction d'activation. Elle définie comme suit :

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

Voici l'allure de la courbe pour f :

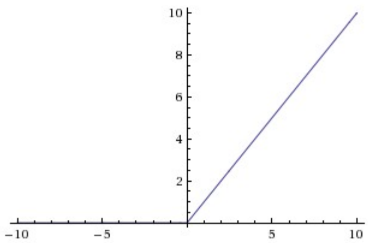


FIGURE 1.4 – Modèle de perceptron multicouche

1.2 Apprentissage

1.2.1 L'apprentissage

Après avoir créer le réseau de neurones, on doit procéder à son apprentissage.

Définition L'apprentissage (en anglais machine learning) est une méthode utilisée en intelligence artificielle. Il s'agit d'algorithmes qui développent la reconnaissance de schémas, l'aptitude à apprendre continuellement et à faire des prévisions grâce à l'analyse d'une base de données.

Dans le domaine des réseaux de neurones, il s'agit d'une phase du développement du réseau durant laquelle le comportement du réseau est modifié jusqu'à l'obtention du comportement désiré. Il y a deux types d'algorithmes d'apprentissage :

1. L'apprentissage supervisé
2. L'apprentissage non supervisé

Dans le cas de l'apprentissage supervisé, les exemples sont des couples (Entrée, Sortie associée à l'entrée) alors que pour l'apprentissage non supervisé, on ne dispose que des valeurs Entrée.

L'apprentissage consiste à modifier le poids des connections entre les neurones. Au démarrage de la phase de l'apprentissage, nous disposons d'une base de données. Nous avons les entrées $(x_i)_{i\in I}$ et les sorties $(\overline{y}_i)_{i\in I}$.

Durant la phase d'apprentissage, nous allons utiliser les entrées $(x_i)_{i\in I}$ connues et tester si l'apprentissage a bien fonctionné en comparant les sorties $(y_i)_{i\in I}$. avec les sorties $(\overline{y}_i)_{i\in I}$ connues de bases.

FIGURE 1.5 – Schéma Entrées/Sorties

Pour que l'apprentissage fonctionne correctement, il est ainsi nécessaire que l'on ait : $y_i \simeq \overline{y_i}\ \forall i \in I$.
Soit f, une fonction paramétrée non linéaire dite d'activation, telle que :

$$y_i = f(x_i, w) = f_i(w)$$

où w est le vecteur poids représentant les paramètres. Plus généralement, on a : $y = f(x, w)$. La sortie y est ainsi fonction non linéaire d'une combinaison des variables x_i pondérées par les paramètres w_i.
On a alors le schéma suivant :

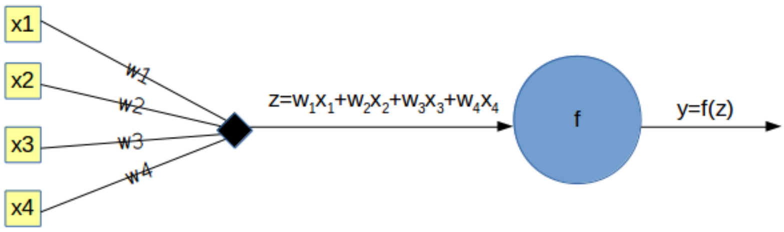


FIGURE 1.6 – Schéma d'un neurone avec 4 entrées

Légende : les carrés jaunes correspondent aux entrées, le losange noir correspond à un nœud et le cercle bleu à un neurone.

Pour que l'apprentissage fonctionne, il suffit alors d'avoir : $\forall i \in I$

$$f_i(w) \simeq \overline{y_i}$$

Le système étant non linéaire, il n'est pas possible d'utiliser les méthodes classiques pour la résolution de systèmes linéaires comme la méthode de Gauss.

Problème Nous cherchons à trouver les éléments du vecteur poids w afin que $\forall i \in I\ f_i(w)$ soit le plus proche possible de $\overline{y_i}$ en utilisant une méthode de résolution de systèmes non linéaires. L'apprentissage est ainsi un problème numérique d'optimisation. Les poids ont initialement des valeurs aléatoires et sont modifiés grâce à un algorithme d'apprentissage.
Par la méthode des moindres carrés, le problème en utilisant la norme 2 se ramène à :

$$f_i(w) \simeq \overline{y_i} \Leftrightarrow \underset{w}{min}(\sum_{i\in I}(f_i(w) - \overline{y_i})^2)$$

Il est aussi possible d'utiliser les normes $\|\cdot\|_\infty$ ou $\|\cdot\|_1$. La fonction de coût des moindres carrés, en ajoutant un coefficient $\frac{1}{2}$ pour simplifier les futurs calculs du gradient, est alors :

$$J(w) = \frac{1}{2} \cdot \sum_{i\in I}(f_i(w) - \overline{y_i})^2$$

1.2.2 Estimation des paramètres d'un réseau de neurones

1.2.2.1 Évaluation du gradient par rétro-propagation

On rappelle que l'objectif est de minimiser la fonction coût des moindres carrées. Le modèle n'étant pas linéaire, il faut avoir recours à des méthodes itératives issues de techniques d'optimisation non linéaire qui modifient les paramètres du modèle en fonction du gradient de la fonction de coût par rapport à ses paramètres. A chaque étape du processus d'apprentissage, il faut évaluer le gradient de la fonction de coût J et modifier les paramètres en fonction de ce gradient afin de minimiser la fonction J. L'évaluation du gradient de la fonction de coût peut être évalué grâce à l'algorithme de rétro-propagation. Nous allons expliquer cette méthode d'évaluation du gradient. Dans cette section, nous n'allons pas différencier les variables selon les couches. Cependant, dans les sections suivantes, nous allons différencier les variables selon les couches afin de simplifier la notation et le stockage de ces variables.

Soit un réseau de neurones à propagation avant avec des neurones cachés et un neurone de sortie. Nous allons changer la définition de la fonction f pour simplifier les notations mais cela ne modifie pas la valeur de la sortie y_i. Ainsi la sortie y_i du neurone i est défini à présent de la manière suivante :

$$y_i = f(\nu_i) = f(\sum_{j=1}^{n_i} w_{ij} x_j^i)$$

avec

- x_j^i la variable j du neurone i. Elle désigne soit la sortie $y_j$du neurone i ou soit une variable d'entrée du réseau.
- n_i le nombre de variables du neurone i. Ces variables peuvent être les sorties d'autres neurones ou les variables du réseau.
- w_{ij} est le poids de la variable j du neurone i.
- ν_i est le potentiel du neurone i.
- f est la fonction d'activation.

Soit l'entier N égal au nombre d'exemples que comprend la phase d'apprentissage. Soit $\overline{y_k}$ la sortie du réseau de neurones pour le $k^{ème}$exemple, elle est appelée la prédiction du modèle pour l'exemple k. La fonction de coût est alors :

$$\begin{aligned} J(w) &= \frac{1}{2} \cdot \sum_{k=1}^{N} (f(\nu_k) - \overline{y_k})^2 \\ &= \frac{1}{2} \cdot \sum_{k=1}^{N} (y_k - \overline{y_k})^2 \end{aligned}$$

avec y_k la valeur prise par la grandeur à modéliser pour l'exemple k.
On pose la fonction de perte relative à l'exemple k $\Pi(x_k, w) = (f(\nu_k) - \overline{y_k})^2 = (y_k - \overline{y_k})^2$ et on a alors :

$$J(w) = \frac{1}{2} \cdot \sum_{k=1}^{N} \Pi(x_k, w)$$

En remarquant que la fonction de perte dépend des variables poids seulement par le potentiel, calculons les dérivées partielles de la fonction Π par rapport aux poids :

$$\begin{aligned} \left(\frac{\partial \Pi(x,w)}{\partial w_{ij}}\right)_{x=x_k} &= \left(\frac{\partial \Pi(x,w)}{\partial \nu_i}\right)_{x=x_k} \cdot \left(\frac{\partial \nu_i}{\partial w_{ij}}\right)_{x=x_k} \\ &= \delta_k^i(x) \cdot \left(\frac{\partial \left(\sum_{l=1}^{n_i} w_{il} x_l^i\right)}{\partial w_{ij}}\right)_{x=x_k} \\ &= \delta_k^i(x) \cdot x_{j,k}^i \end{aligned}$$

avec

- $\nu_i = \sum_{j=1}^{n_i} w_{ij} x_j^i$

— On pose $\delta_k^i(x) = \left(\frac{\partial \Pi(x,w)}{\partial \nu_i}\right)_{x=x_k}$ pour le neurone i pour l'exemple k.

— $x_{j,k}^i$ est la valeur de la variable j du neurone i pour l'exemple k. Ces valeurs sont, à chaque étape du processus d'apprentissage, connues.

Nous cherchons alors à calculer les quantités $\delta_k^i(x)$.

1. Pour le neurone de sortie s de potentiel ν_s,

$$\begin{aligned}
\delta_k^s(x) &= \left(\frac{\partial \Pi(x,w)}{\partial \nu_s}\right)_{x=x_k} \\
&= \left(\frac{\partial}{\partial \nu_s}\left[(f(\nu_k) - \overline{y_k})^2\right]\right)_{x=x_k} \\
&= 2 \cdot (f(\nu_k) - \overline{y_k}) \cdot \left(\frac{\partial f(\nu_s)}{\partial \nu_s}\right)_{x=x_k} \\
&= 2 \cdot (f(\nu_k) - \overline{y_k}) \cdot f'(\nu_s^k)
\end{aligned}$$

   Généralement, la dernière couche est constituée d'un seul neurone muni de la fonction d'activation identité tandis que les autres neurones des couches cachées sont munis de la fonction sigmoïde. C'est un choix arbitraire et cela ne modifie pas les résultats si la couche de sortie a plusieurs de neurones. On considère alors que le neurone de sortie est linéaire et ainsi :

$$\begin{aligned}
\left(\frac{\partial f(\nu_s)}{\partial \nu_s}\right)_{x=x_k} &= \left(\frac{\partial f\left(\sum_{j=1}^{n_s} w_{sj} x_j^s\right)}{\partial \nu_s}\right)_{x=x_k} \\
&= \left(\sum_{j=1}^{n_s} w_{sj} \cdot \frac{\partial f\left(x_j^s\right)}{\partial \nu_s}\right)_{x=x_k} \\
&= \left(\frac{\partial \sum_{j=1}^{n_s} w_{sj} \cdot x_j^s}{\partial \nu_s}\right)_{x=x_k} \\
&= \left(\frac{\partial v_s}{\partial \nu_s}\right)_{x=x_k} \\
&= 1
\end{aligned}$$

   Ainsi, nous obtenons : $\delta_k^s(x) = 2 \cdot (f(\nu_k) - \overline{y_k})$ pour le neurone de sortie s pour l'exemple k.

2. Pour un neurone caché i de potentiel ν_i : la fonction de coût dépend du potentiel ν_i seulement par l'intermédiaire des potentiels des neurones $m \in M \subset I$ dont une des variables est la valeur de la sortie du neurone i, c'est-à-dire $f(\nu_i)$. Cela concerne alors tous les neurones qui sont adjacents au neurone i, entre ce dernier neurone et la sortie, sur le graphe du réseau de neurones (voir le schéma ci-dessous).

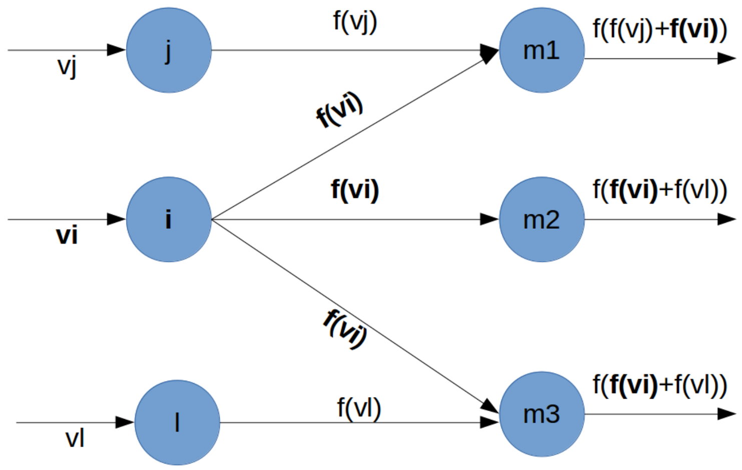


FIGURE 1.7 – Schéma des neurones m et i (sans les poids)

$$\begin{aligned}
\delta_k^i(x) &= \left(\frac{\partial \Pi(x,w)}{\partial \nu_i}\right)_{x=x_k} \\
&= \sum_{m \in M} \left(\left(\frac{\partial \Pi(x,w)}{\partial \nu_m}\right)_{x=x_k} \cdot \left(\frac{\partial \nu_m}{\partial \nu_i}\right)_{x=x_k} \right) \\
&= \sum_{m \in M} \left(\delta_k^m(x) \cdot \left(\frac{\partial \nu_m}{\partial \nu_i}\right)_{x=x_k} \right) \\
&= \sum_{m \in M} \left(\delta_k^m(x) \cdot \left(\frac{\partial \left(\sum_{j=1}^{n_m} w_{mj} x_j^m \right)}{\partial \nu_i} \right)_{x=x_k} \right) \\
&= \sum_{m \in M} \left(\delta_k^m(x) \cdot \left(\frac{\partial \left(\sum_{i=1}^{n_m} w_{mi} \cdot f(v_i) \right)}{\partial \nu_i} \right)_{x=x_k} \right) \\
&= \sum_{m \in M} \left(\delta_k^m(x) \cdot w_{mi} \cdot f'(v_i^k) \right) \\
&= f'(v_i^k) \cdot \sum_{m \in M} \left(\delta_k^m(x) \cdot w_{mi} \right)
\end{aligned}$$

On peut ainsi remarquer que $\delta_k^i(x)$ peuvent se calculer de manière récursive, c'est-à-dire en parcourant le graphe de la sortie vers l'entrée du réseau : c'est la rétro-propagation.

Ainsi nous pouvons calculer le gradient de la fonction de coût, comme suit :

$$\frac{\partial J(w)}{\partial w} = \frac{1}{2} \cdot \sum_{k=1}^{N} \frac{\partial \Pi(x_k, w)}{\partial w}$$

1.2.2.2 Résumé de la rétro-propagation

Résumons les différentes étapes de la retro-propagation :

1. La propagation avant : les variables de l'exemple k sont utilisées pour calculer les sorties et les potentiels de tous les neurones.
2. La retro-propagation : les quantités $\delta_k^i(x)$ sont calculés récursivement.

$$\delta_k^i(x) = f'(v_i^k) \cdot \sum_{m \in M} \left(\delta_k^m(x) \cdot w_{mi} \right)$$

Pour le neurone de sortie, on a : $\delta_k^s(x) = 2 \cdot (f(\nu_k) - \overline{y_k})$.

3. Calcul du gradient des fonctions de perte : $\left(\dfrac{\partial \Pi(x,w)}{\partial w_{ij}}\right)_{x=x_k} = \delta_k^i(x) \cdot x_{j,k}^i$.
4. Calcul du gradient de la fonction de coût : $\dfrac{\partial J(w)}{\partial w} = \dfrac{1}{2} \cdot \sum_{k=1}^{N} \dfrac{\partial \Pi(x_k, w)}{\partial w}$.

Nous sommes ainsi capable d'évaluer le gradient de la fonction de coût, à chaque itération de l'apprentissage, par rapport aux paramètres du modèle que sont les poids. Il suffit, à présent, de modifier les paramètres du modèle afin de minimiser cette fonction de coût et de définir un critère d'arrêt pour la minimisation du gradient de la fonction de coût.

1.2.2.3 Modification des poids

La règle delta, appelé méthode du gradient simple, stipule que :

$$w_{ij} = w_{ij} - \eta_i \cdot \delta_k^j(x) \cdot f(\nu_i^k)$$

avec $\eta_i > 0$ un scalaire, appelé pas d'apprentissage ou pas de gradient qui peut être fixé ou adaptatif. Ce pas d'apprentissage est très important et aura une influence sur la convergence de la solution. Plus ce pas est petit et plus la convergence sera lente. Et plus ce pas est grand et plus la solution aura tendance à diverger. Il faut alors judicieusement choisir le pas. Nous définirons, par la suite, comment le définir.

1.3 L'algorithme d'apprentissage par rétro-propagation

1.3.1 Notations

Résumons les différentes étapes à suivre pour l'apprentissage. Pour cela, nous allons construire un algorithme. Afin de simplifier les calculs et la compréhension de l'algorithme, quelques notations seront modifiées. Dans cet algorithme, on considère que la couche de sortie est composée d'un ou de plusieurs neurones. On part du principe que les neurones sont tous reliés entre eux par un poids (qui est égale 0 si la liaison n'existe pas réellement). De plus, on considère que les couches n'ont pas forcément le même nombre de neurones et on suppose que les données d'entrées sont de même taille pour tous les exemples. Nous considérons aussi que les neurones n'ont pas de biais. Pour finir, nous allons aussi différencier les variables selon la couche auxquelles elles appartiennent.

En utilisant les calculs des sections précédentes, on pose :

K : nombre d'exemples à disposition avec $K \in \mathbb{N}^*$

L : nombre de couches avec $L \in \mathbb{N}^*$et $L \geq 2$.

n_i : le nombre de neurones pour la couche i avec $i \in \mathbb{N}^* et\ i \in [\![1, L]\!]$

$\mathrm{N} = \underset{i=1..L}{max}(n_i)$: le nombre maximal de neurones par couche avec $N \in \mathbb{N}^*$.

w_{ij}^l : le poids qui relie le neurone i de la couche l au neurone j de la couche l+1avec $i \in [\![1, n_l]\!]$et j$\in [\![1,n_{l+1}]\!]$et $l \in [\![1, L-1]\!]$

$a_i^l = f(\nu_i^l)$: la valeur dit aussi activité du ième neurone de la couche l.

$x_i = a_i^1$: la ième valeur du neurone de la couche d'entrée.

$y_i = a_i^L$: la ième valeur du neurone de la couche de sortie.

$\nu_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^{l-1} a_i^{l-1}$: le potentiel du neurone j de la couche l.

$\delta_j^L = 2(a_j^L - \overline{y_j})$: la valeur du gradient pour le neurone j de la couche de sortie.

$\delta_j^l = f'(\nu_j^l) \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^l$ la valeur du gradient pour le neurone j de la couche l.

$J_k = \frac{1}{2} \sum_{i=1}^{n_L} (y_i - \overline{y_i})_{x=x_k}^2$: l'erreur quadratique entre la sortie obtenue et la sortie attendue pour l'exemple k.

$\Gamma = \underset{k \in K}{max}(J_k)$: la valeur maximale entre tous les J_k.

ε : la précision souhaitée, une valeur réelle très proche de 0.

η_l : le pas d'apprentissage pour la couche l.

1.3.2 L'algorithme

Algorithme 1.1 Algorithme de rétro-propagation

Entrées: Un ensemble d'exemples avec comme vecteur entrée x et comme vecteur sortie y. Et un réseau avec un nombre de couches et un nombre de neurones pré-définis. Et un ϵ pour la précision définie.
Sorties: Un réseau de neurones avec des poids.

Pour l=1 à L-1 **faire**
  Pour chaque poids w_{ij}^l **faire**
    $w_{ij}^l \leftarrow$ valeur aléatoire relativement petite
  Finpour
Finpour
Répéter
  Pour chaque exemple $(x, \overline{y})_k$ k=1..K **faire**
    Pour i=1 à n_1 **faire**
      $a_i^1 \leftarrow x_i$
    Finpour
    Pour l=2 à L **faire**
      Pour j=1 à n_l **faire**
        $\nu_j^l \leftarrow \sum\limits_{i=1}^{n_{l-1}} w_{ij}^{l-1} a_i^{l-1}$
        $a_j^l \leftarrow f(\nu_j^l)$
      Finpour
    Finpour
    Pour j=1 à n_L **faire**
      $\delta_j^L \leftarrow 2(a_j^L - \overline{y_j})$
    Finpour
    Pour l=L-1 à 2 **faire**
      Pour j=1 à n_l **faire**
        $\delta_j^l \leftarrow f'(\nu_j^l) \sum\limits_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^l$
      Finpour
    Finpour
    Pour l=1 à L-1 **faire**
      Pour chaque poids w_{ij}^l **faire**
        $w_{ij}^l \leftarrow w_{ij}^l - \eta_l \cdot \delta_j^l(x) \cdot a_i^l$
      Finpour
    Finpour
    $J_k \leftarrow \frac{1}{2} \sum\limits_{i=1}^{n_L} \left(a_i^L - \overline{y_i}\right)^2$
  Finpour
  $\Gamma \leftarrow \| J \|_\infty$
Jusqu'à $\Gamma \leq \varepsilon$

1.3.3 Matrices de stockage

Nous avons besoin de stocker les différentes valeurs calculées. Nous choisissons de les stocker dans des matrices définies ci-dessous. Ce choix est arbitraire et ne définit la manière dont nous allons stocker nos données lors de la programmation de notre réseau de neurones.

— $X = \begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^K \\ x_2^1 & x_2^2 & \cdots & x_2^K \\ \vdots & \vdots & \cdots & \vdots \\ x_{n_1}^1 & x_{n_1}^2 & \cdots & x_{n_1}^K \end{pmatrix} \in \mathbb{R}^{n_1 \times K}$ définit la matrice des données d'entrées. Chaque colonne correspond à un exemple.

— $\overline{Y} = \begin{pmatrix} \overline{y}_1^1 & \overline{y}_1^2 & \cdots & \overline{y}_1^K \\ \overline{y}_2^1 & \overline{y}_2^2 & \cdots & \overline{y}_2^K \\ \vdots & \vdots & \cdots & \vdots \\ \overline{y}_{n_L}^1 & \overline{y}_{n_L}^2 & \cdots & \overline{y}_{n_L}^K \end{pmatrix} \in \mathbb{R}^{n_L \times K}$ définit la matrice des données de sorties. Chaque colonne correspond à un exemple.

— $W_{l\in[\![1,L-1]\!]} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n_{l+1}} \\ w_{21} & w_{22} & \cdots & w_{2n_{l+1}} \\ \vdots & \ddots & \cdots & \vdots \\ \vdots & \vdots & \ddots & w_{n_l-1,n_{l+1}} \\ w_{n_l1} & \cdots & w_{n_ln_{l+1}-1} & w_{n_ln_{l+1}} \end{pmatrix}_l \in \mathbb{R}^{n_l\times n_{l+1}}$ avec w_{ij} poids qui relie le neurone i de la couche l au neurone j de la couche l+1. L'ensemble de ces matrices définissent l'ensemble des poids du réseau.

— $A = \begin{pmatrix} a_1^1 & a_1^2 & \cdots & a_1^L \\ a_2^1 & a_2^2 & \cdots & a_2^L \\ \vdots & \vdots & \cdots & \vdots \\ a_N^1 & a_N^2 & \cdots & a_N^L \end{pmatrix} \in \mathbb{R}^{N\times L}$ la matrice de l'ensemble des activités de tous les neurones du réseau avec a_i^j correspondant à l'activité du neurone i pour la couche j. La dernière colonne correspond aux valeurs de sorties de l'apprentissage qu'il faut comparer avec les données de sorties de départ. Prenant en compte que chaque couche n'a pas forcément le même nombre de neurones, a_i^j prend la valeur de 0 si le neurone i n'existe pas.

— $\gamma = \begin{pmatrix} \nu_1^2 & \nu_1^3 & \cdots & \nu_1^L \\ \nu_2^2 & \nu_2^3 & \cdots & \nu_2^L \\ \vdots & \vdots & \cdots & \vdots \\ \nu_N^2 & \nu_N^3 & \cdots & \nu_N^L \end{pmatrix} \in \mathbb{R}^{N\times L}$ la matrice de l'ensemble des potentiels de tous les neurones du réseau avec ν_i^j correspondant au potentiel du neurone i pour la couche j. Pour les mêmes raisons que précédemment, ν_i^j prend la valeur de 0 si le neurone i n'existe pas.

— $\Delta = \begin{pmatrix} \delta_1^2 & \delta_1^3 & \cdots & \delta_1^L \\ \delta_2^2 & \delta_2^3 & \cdots & \delta_2^L \\ \vdots & \vdots & \cdots & \vdots \\ \delta_N^2 & \delta_N^3 & \cdots & \delta_N^L \end{pmatrix} \in \mathbb{R}^{N\times L}$ la matrice de l'ensemble des gradients pour tous les neurones du réseau avec δ_i^j correspondant au gradient du neurone i pour la couche j. Pour les mêmes raisons que précédemment, δ_i^j prend la valeur de 0 si le neurone i n'existe pas.

— $J = \begin{pmatrix} J_1 \\ J_2 \\ \vdots \\ J_K \end{pmatrix} \in \mathbb{R}^K$ l'ensemble des erreurs pour tous les exemples avec J_i correspondant à l'erreur pour l'exemple i.

1.3.4 Exemple d'un apprentissage

Nous allons appliquer l'algorithme précédent afin de mieux comprendre son fonctionnement. Nous allons utilisé un réseau de neurones avec L=4 couches comprenant une couche d'entrée et de sorties ainsi que deux couches cachées. La couche d'entrée ainsi que les couches cachées contiennent chacune deux neurones. La couche de sortie contient un seul neurone. Nous allons utilisé qu'un seul exemple. Voici les données :

$$\text{Entrées } X = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

$$\text{Sortie } \overline{Y} = (1)$$

— On a N=2, L=4 et $n_1 = n_2 = n_3 = 2$ et $n_4 = 1$.

— De plus, on a initialisé aléatoirement les poids de la sorte :

$$W_1 = \begin{pmatrix} 1 & 2 \\ 0.5 & -1 \end{pmatrix}$$

$$W_2 = \begin{pmatrix} -1 & 2 \\ 1 & 3 \end{pmatrix}$$

$$W_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

— On prend comme précision $\epsilon = 10^{-6}$ et on prend un pas d'apprentissage identique pour chaque couche : $\eta = 0.1$.

— La fonction d'activation est $f(x) = \frac{1}{1+e^{-x}}$ et $f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)\cdot(1-f(x))$.

1ere étape En appliquant l'algorithme pour la propagation avant, on obtient :

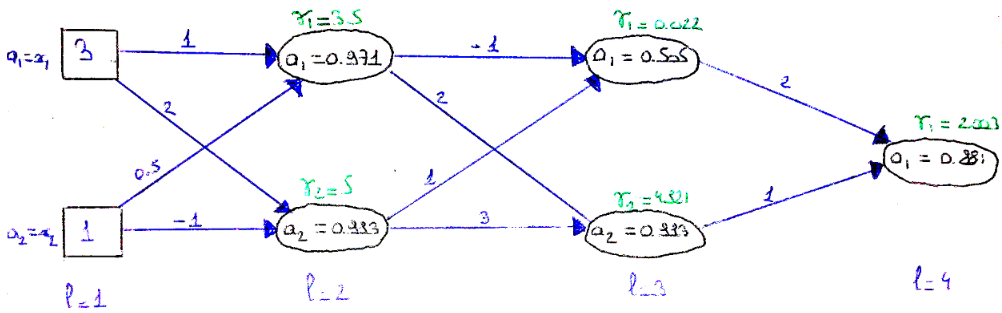


FIGURE 1.8 – Exemple propagation avant

Et alors on a :

$$A = \begin{pmatrix} 3 & 0.971 & 0.505 & 0.881 \\ 1 & 0.993 & 0.993 & 0 \end{pmatrix}$$
$$\gamma = \begin{pmatrix} 3.5 & 0.022 & 2.003 \\ 5 & 4.921 & 0 \end{pmatrix}$$

et $J_1 = \frac{1}{2}(0.881 - 1)^2 = 0.007$.
Exemple : $\nu_1^1 = 1 \cdot 3 + 0.5 \cdot 1 = 3.5$ et $a_1^1 = f(3.5) = 0.971$.

2ème étape En appliquant l'algorithme pour la propagation arrière, on obtient :

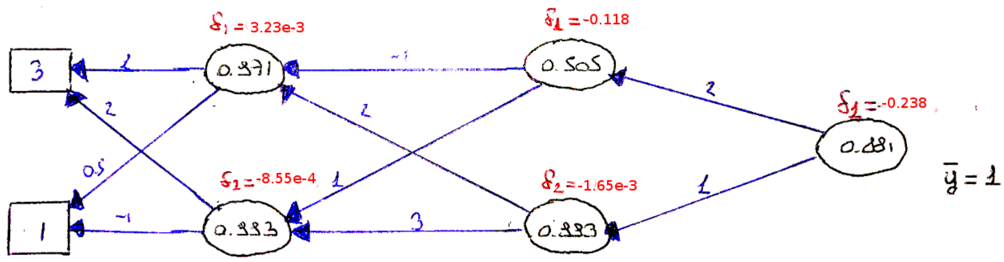


FIGURE 1.9 – Exemple propagation arrière

Et alors on a : $\gamma = \begin{pmatrix} 3.23 \cdot 10^{-3} & -0.118 & -0.238 \\ -8.55 \cdot 10^{-4} & -1.65 \cdot 10^{-3} & 0 \end{pmatrix}$.
Exemple : $\delta_1^1 = f'(\nu_1^1) \cdot (-1 \cdot (-0.118) + 2 \cdot (-1.65 \cdot 10^{-3})) = f(\nu_1^1) \cdot (1 - f(\nu_1^1)) \cdot 0.1147 = 0.971 \cdot (1 - 0.971) \cdot 0.1147 = 3.23 \cdot 10^{-3}$.

3ème étape En modifiant les poids, on obtient :

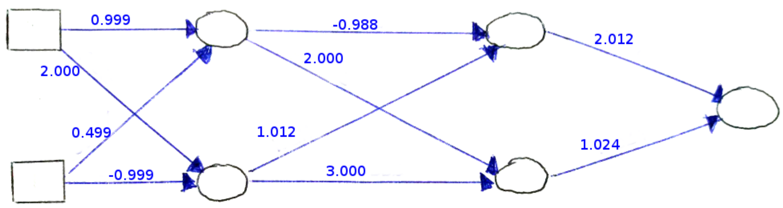


FIGURE 1.10 – Exemple modification des poids

et on a alors : $W_1 = \begin{pmatrix} 0.999 & 2.000 \\ 0.490 & -0.999 \end{pmatrix}$ et $W_2 = \begin{pmatrix} -0.988 & 2.000 \\ 1.012 & 3.000 \end{pmatrix}$ et $W_3 = \begin{pmatrix} 2.012 \\ 1.024 \end{pmatrix}$.
On peut ainsi recommencer l'apprentissage avec un autre exemple et avec ces nouveaux poids initiaux. Ayant un seul exemple, on a alors $\Gamma = J_1 < \varepsilon$ et il faut alors continuer l'apprentissage.
Exemple : $w_{11}^1 = 1 - 0.1 \cdot 3 \cdot 3.23 \cdot 10^{-3} = 0.999$.

Chapitre 2

Composition musicale : les données

2.1 Les fichiers MIDI

Un fichier MIDI (Musical Instrument Digital Interface), contrairement au fichier audio, ne contient aucun son, à proprement parler, mais une série de « directives » que seul un instrument compatible MIDI peut comprendre. L'instrument MIDI d'après les « consignes » contenues dans le fichier MIDI peut alors produire le son. Les fichiers MIDI sont des fichiers binaires : les données sont stockées en format binaire et non codées (texte, objets etc...). Vous trouverez en annexe des explications plus détaillées sur ces fichiers MIDI.

Nous avions choisi d'étudier un seul genre de musique afin de rendre l'apprentissage plus simple et plus précis. Pour cela, nous avons trouvé sur Internet des fichiers MIDI contenant des musiques classiques jouées au piano (`http://www.piano-midi.de/`). Nous avons alors à notre disposition **336 fichiers MIDI**.

Chaque fichier MIDI représente une musique créée par un musicien. Toutes les musiques ont des durées et des notes différentes. Les éventements MIDI sont aussi différents selon les musiques.

DEFINITION NoteOnEvent, tick, data, channel, track

2.2 L'extraction d'informations

Notre objectif est d'obtenir deux fichiers de données : l'un pour l'apprentissage et l'un pour les tests. Dans le fichier se trouvera toutes les informations que l'on jugera nécessaire pour la création d'une musique. Et ainsi dans un fichier se trouvera plusieurs musiques avec leurs différentes informations.

2.2.1 Script Python

Tout d'abord, nous avons construit un programme python permettant d'extraire des informations d'un fichier MIDI. Cependant, étant donné que les fichiers MIDI sont très complexes et très difficiles à manipuler, nous avons finalement décidé d'utiliser une bibliothèque python permettant de faire cela. Cette bibliothèque est appelé « python-midi » et peut être trouvée à l'adresse suivante : `https://github.com/vishnubob/python-midi`. Il s'agit d'un projet mis sur GitHub.

Cette bibliothèque est très intéressante car elle permet de créer un script Python à partir d'un fichier MIDI. Par exemple, si nous avons le fichier midi suivant : « alb_esp1.mid » et que nous voulons voir les différents évenements qui permettent de créer la musique de ce fichier, il suffit d'utiliser la commande suivante :

mididump.py mary.mid

Et nous obtenons alors le script (raccourci) suivant :

```
midi.Pattern(format=1, resolution=480, tracks=\
[midi.Track(\
  [midi.TrackNameEvent(tick=0, text='Espana␣Op.␣165', data=[69, 115, 112, 97,
      110, 97, 32, 79, 112, 46, 32, 49, 54, 53]),
...
   midi.SetTempoEvent(tick=240, data=[7, 113, 125]),
...
   midi.EndOfTrackEvent(tick=0, data=[])]),
 midi.Track(\
```

```
  [midi.TrackNameEvent(tick=0, text='Piano␣right', data=[80, 105, 97, 110, 111,
     32, 114, 105, 103, 104, 116]),
   midi.ProgramChangeEvent(tick=0, channel=0, data=[0]),
   midi.ControlChangeEvent(tick=0, channel=0, data=[7, 100]),
   midi.NoteOnEvent(tick=160, channel=0, data=[64, 0]),
   midi.NoteOnEvent(tick=160, channel=0, data=[62, 0]),
...
```

Nous pouvons aussi faire l'opération dans l'autre sens en créant un fichier MIDI à partir d'un script Python. Pour cela, il suffit d'utiliser le Pattern crée par le script et d'utiliser la méthode « midi.write_file » de la manière suivante :

midi.write_midifile("newMusique.mid", pattern)')

Par cette commande, nous obtenons dans le répertoire courant un fichier MIDI que l'on peut écouter.

Conclusion Nous avons à présent la possibilité de créer un fichier MIDI à partir d'un script python et réciproquement. Maintenant, nous avons comme objectif de créer ces scripts pour tous les fichiers que l'on dispose afin d'extraire, par la suite, les informations que l'on souhaite.

2.2.2 Analyse des scripts

Nous avons besoin d'extraire des informations de types identiques pour chacune de nos données. Ainsi, nous avons lu la plupart de nos fichiers afin d'en déduire les informations communes entre nos données mais aussi la façon dont est écrit les fichiers afin de pouvoir lire automatiquement et identiquement tous nos fichiers. Nous avons alors détecter que tous les fichiers ont la même entête et qu'ils ont au moins 2 « tracks » (pistes). De plus, les événements « NoteOnEvent » sont contiennent toujours le même nombre d'arguments comme suit :

midi.NoteOnEvent(tick=160, channel=0, data=[62, 0])

Chaque track finisse de la même façon et toutes les fichiers se terminent de la même manière. Les événements différents de NoteOnEvent sont optionnels et n'importent pas d'informations essentiels pour joueur des notes. Nous avons alors décidé de récupérer les arguments pour tous les évenements NoteOnEvent de chaque fichier que l'on dispose.

La valeur de « channel » est toujours nulle et ainsi nous ne tiendrons pas en compte cet argument. Nous prenons alors :

1. Tick, représentant le temps.
2. Data1, représentant la note.
3. Data2, représentant la velocity.

Conclusion Nous avons décidé de ne prendre en compte que les événements NoteOnEvent ainsi que les 3 valeurs les représentant. Nous allons alors devoir créer un programme qui extrait les 3 valeurs pour tous les événements NoteOnEvent pour tous nos fichiers.

2.2.3 Extraction des informations

Vous trouverez le programme nommé **« creation_donnees.py »** permettant d'extraire les informations en annexe. Ce programme permet d'abord de créer un script Python complet pour tous nos fichiers MIDI. Ensuite, il crée un deuxième script simplifié pour chacun de ses scripts, c'est-à-dire que ces nouveaux scripts ne contiennent seulement les événements NoteOnEvent ainsi que seulement 2 tracks. Puis, à partir de ces scripts, le programme permet de créer des fichiers textes pour chaque script. Chaque fichier texte représente alors une musique et il y a autant de lignes qu'il y a d'événements NoteOnEvent. Chaque ligne contient les trois valeurs citées précédemment. Les noms des fichiers sont les titres de la musique qu'ils représentent. Par exemple, si on a le fichier MIDI « alb_esp1.txt » :

1. D'abord, le script complet est créé : « alb_esp1fMidiComplet.py ».
2. Ensuite le script simplifié est créé : « alb_esp1fMidiSimple.py ».
3. Enfin, le fichier teste est créé : « alb_esp1.txt » qui commence de la sorte :

240	81	60
240	81	81
0	88	66
..	..	..

2.3 Analyse de nos données

2.3.1 Les données initiales

Nous avons maintenant 336 fichiers textes représentant chacun une musique unique et des nombres de lignes différents ainsi que des valeurs différentes selon les fichiers. Pour bien effecteur l'apprentissage, nous avons besoin d'analyser nos données.

Objectif Le but est d'avoir le même nombre de notes pour les données de l'apprentissage et le même nombre de notes pour les notes du test. Aussi, nous devons éliminer les valeurs qui sont trop élevées afin que l'on puisse faire une normalisation efficace par la suite.

Analyse Vous trouverez le programme « analyse.py » qui nous a permis d'analyser nos données. Nous avons cherché par différents tests, les valeurs pour le « tick » et pour le nombre de lignes optimales afin de ne perdre le moins d'informations possibles. Aussi, nous voulions respecté le principe des 2/3 de données pour l'apprentissage et 1/3 des données pour le test.

Voici les résultats de l'analyse pour le nombre de lignes et pour les valeurs des ticks :

Nombre de lignes	Maximum	31 900
	Minimum	334
	Moyenne	4 621
	Médiane	3 247
	supérieur à 2000 (inclus)	242
	compris entre à 440(inclus) et 2000	93
Tick	Maximum	155 520
	Minimum	0
	Moyenne	85.7
	Médiane	0
	Écart-type	317.23
	supérieur à 3800	289
	inférieur à 3800 (inclus)	1 552 563

En choisissant un nombre de lignes égale à 2000 pour l'apprentissage et un nombre de lignes égale à 440 pour les tests, nous obtenons 242 musiques pour l'apprentissage et 93 pour le test. Nous constatons que les valeurs des ticks sont très éparpillés et qu'il y a quelques valeurs aberrantes. En effet, seulement 289 ticks sont au dessus de 3800 alors que 99.9% des ticks sont inférieurs à 3800.

En écartant toutes les notes dont la valeur de « tick » est supérieur à 3800 et en respectant le nombre de lignes, nous obtenons :

Apprentissage	71%	241 musiques
Test	29%	94 musiques
		= 335

Seulement une seule musique sera alors exclue et était doté de seulement 334 notes.

Voici l'analyse pour les deux autres valeurs :

Key	Maximum	107
	Minimum	21
	Moyenne	64.7
	Médiane	65
	Écart-type	13.40
Velocity	Maximum	127
	Minimum	0
	Moyenne	25.4
	Médiane	0
	Écart-type	28.04

2.3.2 Les données partagées

Nous pouvons maintenant analyser les données que nous allons utiliser pour l'apprentissage et pour les tests.

Voici les résultats pour l'apprentissage et le test après avoir retirer les valeurs trop élevés ainsi qu'après le découpage des fichiers pour réduire le nombre de lignes :

		Apprentissage + Test	Apprentissage	Test
Tick	**Maximum**	3780	3780	3720
	Minimum	0	0	0
	Moyenne	84.15	103.8	82.47
	Médiane	0	0	0
	Écart-type	168.49	211.3	164.18
Key	**Maximum**	106	103	106
	Minimum	21	21	22
	Moyenne	70.42	69.0	70.54
	Médiane	71	69.0	71
	Écart-type	10.16	9.02	10.24
Velocity	**Maximum**	127	110	127
	Minimum	0	0	0
	Moyenne	26.55	25.502	26.65
	Médiane	11.0	16.0	0
	Écart-type	28.88	27.46	29.00

Matrice de corrélation Pour l'apprentissage :

Matrice de corrélation:

Variables	tick	key	velocity
tick	**1**	-0,020	-0,263
key	-0,020	**1**	0,088
velocity	-0,263	0,088	**1**

Figure 2.1 – Matrice de corrélation (Apprentissage)

Pour le test :

Variables	tick	key	velocity
tick	**1**	-0,008	-0,272
key	-0,008	**1**	0,090
velocity	-0,272	0,090	**1**

Figure 2.2 – Matrice de corrélation (Test)

Nous pouvons en déduire que les variables ne sont pas du tout corrélées.

Effectif (Key) Pour l'apprentissage :

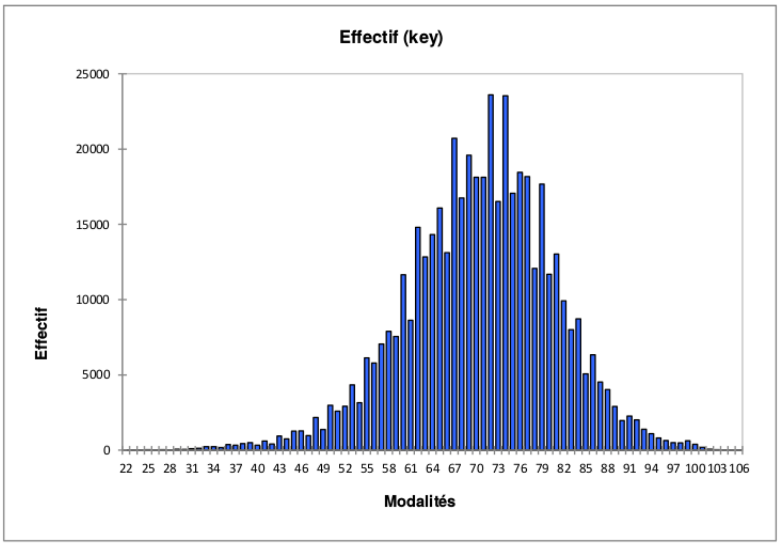


FIGURE 2.3 – Effectif Key (Apprentissage)

Pour le test :

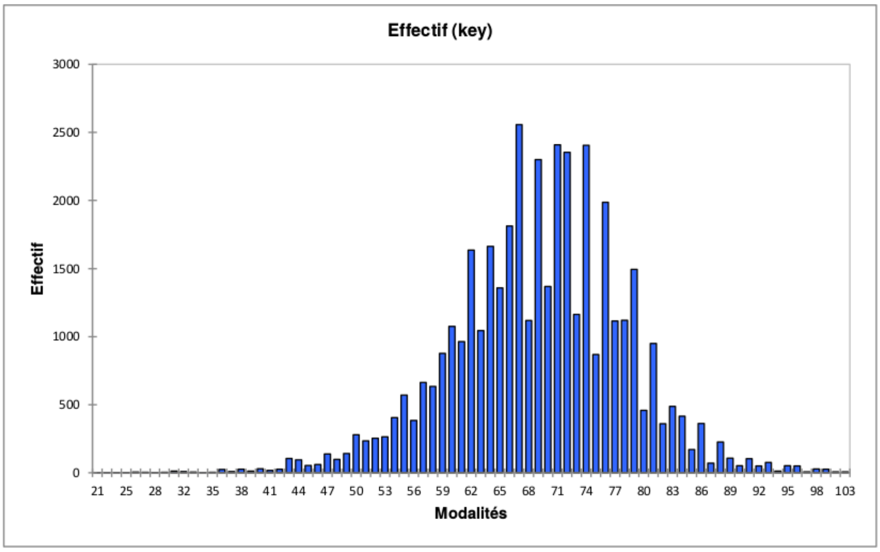


FIGURE 2.4 – Effectif Key (Test)

On peut se rendre compte que peu importe l'échantillon, les « keys » pour les musiques classiques sont très centrées autour de 70.

Effectif (Velocity) Pour l'apprentissage :

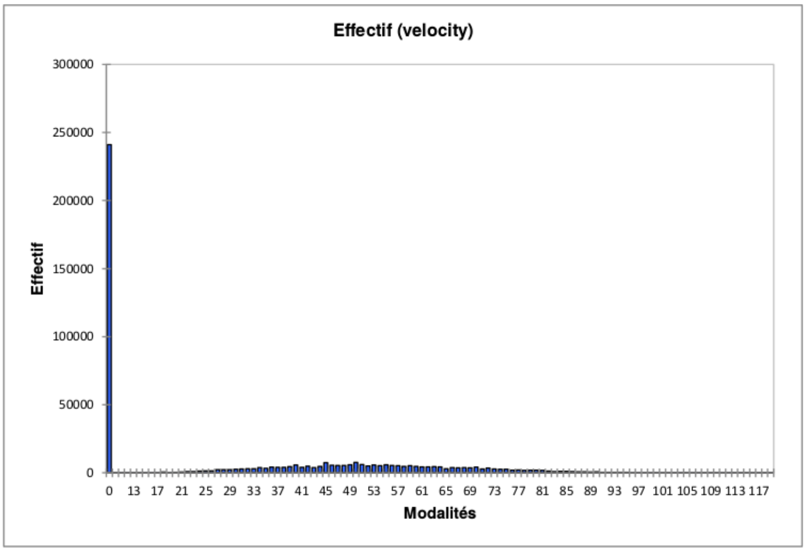


Figure 2.5 – Effectif Velocity (Apprentissage)

Pour le test :

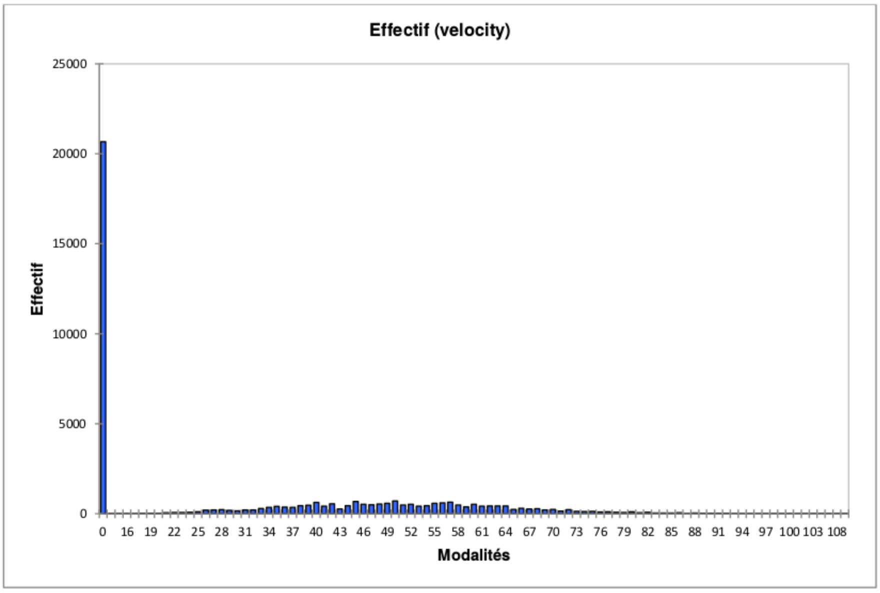


Figure 2.6 – Effectif Velocity (Test)

On remarque alors que les valeurs de la velocity sont relativement éparpillées mais plutôt centrées autour de 50. Cependant nous constatons un pic au niveau du 0 pour les deux échantillons.

Effectif (Tick) Pour l'apprentissage :

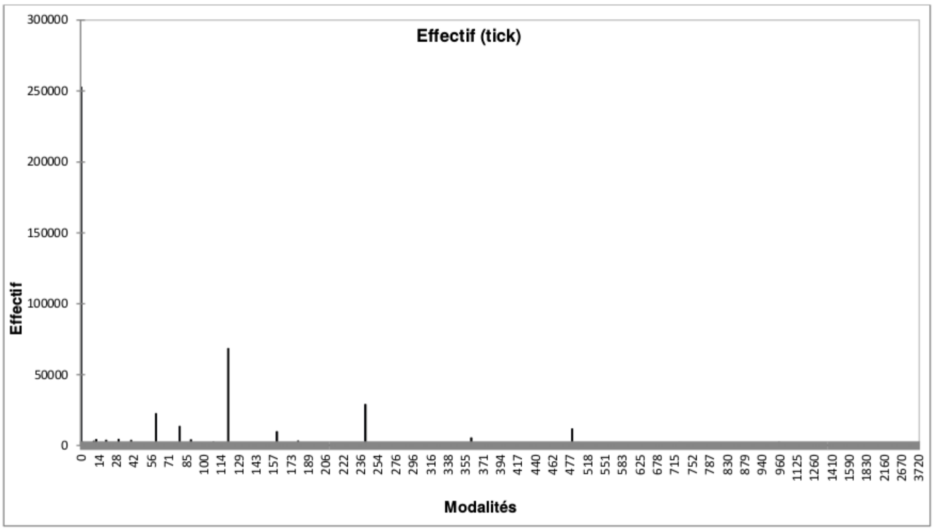


FIGURE 2.7 – Effectif Tick (Apprentissage)

Pour le test :

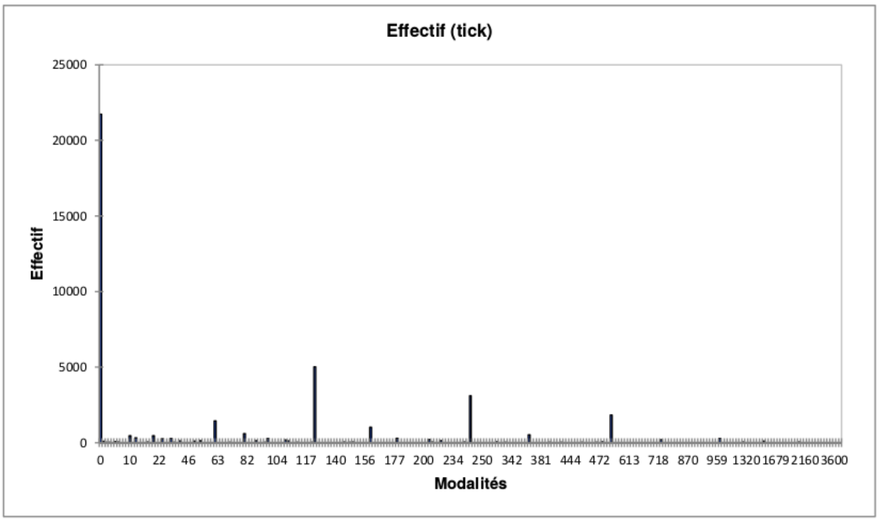


FIGURE 2.8 – Effectif Tick (Test)

On remarque que les valeurs des ticks sont très étalées sur la gamme des valeurs possibles et qu'il n'y a pas vraiment de tendance. On constate tout de même un pic pour la valeur 0.

Conclusion Nous avons remarqué que la musique classique a ses propres caractéristiques. En effet, pour les deux échantillons nous avons retrouvé les mêmes caractéristiques. Cependant les pics pour les effectifs pour la modalité 0 sont à prendre compte car ils pourraient avoir une influence sur le résultat final. Toutes ces remarques vont nous permettent de bien définir notre réseau de neurones ainsi que le choix de la normalisation.

2.4 Normalisation de nos données

Ayant des valeurs largement supérieures à 1, nous allons normaliser nos données sur l'intervalle 0 et 1.
La première normalisation que l'on peut appliquer est la suivante :

$$x'_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

Elle permet de normaliser nos données sur l'intervalle [0,1].

La deuxième qui peut être appliquée est la suivante :

$$x'_i = \frac{1}{2}\left(0.01 \cdot tanh(\frac{x_i - moyenne(x)}{ecartType(x)}) + 1\right)$$

Elle permet aussi de normaliser les données sur l'intervalle [0,1].

Chapitre 3

Composition musicale : le réseau de neurones

Nous allons définir dans cette partie l'architecture de notre réseau. Nous allons programme un réseau de neurone récurrent. Dans cette partie, nous utiliserons le programme « main.py » que vous trouverez en annexe.

3.1 Structure de données

Dans cette section, nous supposons qu'une séquence est de taille 10 mais cette valeur est arbitraire.

Pour l'entrainement, nous souhaitons que le réseau suive le modèle suivant :

1. Le réseau reçoit un tableau de notes en entrée et prend les 10 premières notes.
2. Le réseau prédit la note qui suit les 10 notes passées en entrée (la 11ème note).
3. Le réseau décale l'entrée d'un rang de 1 et prédit la note qui suit la note de l'étape 2.
4. etc... jusqu'à la fin de musique.

Pour cela, nous allons stocker nos données d'entrée dans un tableau de 3 dimensions. La première dimension sera le nombre de séquences de taille 10 possibles par musiques. Nous rappelons que les musiques sont maintenant de même taille, c'est-à-dire qu'elles ont le même nombre de notes. La deuxième dimension représente la taille des séquences, c'est-à-dire 10. Enfin, la troisième dimension sera le nombre de caractéristiques par note, c'est-à-dire 3 (tick, key et velocity). Ensuite, la sortie du réseau sera de dimension 2 avec les mêmes dimensions que les deux premiers de l'entrée.

Tout d'abord, nous avons deux fichiers : l'un contenant toutes les musiques pour l'apprentissage et l'autre pour le test. En utilisant une fonction « creationDonnees » définie dans notre programme, les fichiers sont lus et sont transformés en un tableau de 3 dimensions. Ainsi, nous obtenons :

	Apprentissage			Test		
	D1	D2	D3	D1	D2	D3
X	479 590	10	3	40 420	10	3
Y	479590	3	//	40 420	3	//

avec X le tableau pour l'entrée et Y pour la sortie.

Le réseau de neurones prendra alors une séquence de taille 10 en entrée.

3.2 Le réseau récurrent avec LSTM

3.2.1 Réseau de neurones récurrent

Présentation

Un réseau de neurone standard donne une valeur en sortie à partir d'une donnée en entrée. Cependant une telle représentation ne tient compte que d'une entrée à l'instant présent. Ce type de réseau ne peut donc pas apprendre de schémas ou de suites récurrentes. Or, pour générer de la musique nous avons besoin que le réseau soit capable d'analyser un suite de note et de prévoir la note en suivante en fonction de cette suite et non pas seulement de la dernière note. C'est pourquoi nous avons utilisé un réseau de neurones récurrent.

Dans un réseau de neurones récurrent, l'information peut se propager dans les deux sens. C'est à dire que l'information va des premières couches vers les couches profondes mais l'information sortant des couches

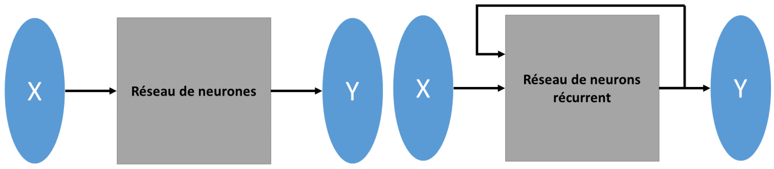


FIGURE 3.1 – Réseau de neurones standard et récurrent

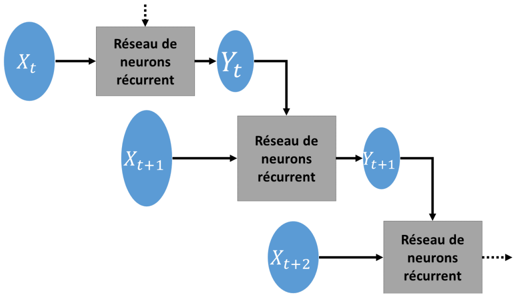


FIGURE 3.2 – Réseau de neurones récurrent développé

profondes peut aussi devenir une entrée pour une première couche. Si l'information provient d'une couche plus profonde il s'agit alors d'une information passée puisqu'elle a été générée par l'entrée précédente (la suite de note précédente). Ainsi dans ce type de réseau une information peut être conservée « en mémoire » en étant réinjectée dans le réseau.

On peut avoir une représentation développée en fonction du temps t :

D'une façon plus générale on peut avoir une sortie du réseau différente de l'entrée qui sera réinjectée :

Un réseau de neurone récurrent standard (aussi appelé « Vanilla RNN » en anglais) peut être représenté ainsi :

Problèmes du réseau de neurones récurrent standard

— Mémoire très court terme

Avec un tel réseau, une information qui serait importante ne sera pas sauvegardée de façon privilégiée par rapport aux entrées suivantes et précédentes. Aussi, cette information sera petit à petit oubliée, ou plus précisément, « diluée » parmis les autres données. Les dernièeres informations auront donc plus de poids.

— Disparition du gradient

Les réseaux de neurones récurrent utilisent les fonctions d'activation sigmoïdes et peuvent donc conduire au problème de disparition du gradient.

En effet, la méthode du gradient fonctionne en analysant le changement sur la sortie impacté par un changement sur certains neurones. Or ce changement peut se trouver très faible à cause des fonctions d'activations sigmoïdes. Par exemple si l'entrée d'une sigmoïde tanh est 2 et qu'on la modifie en 3 on n'observera un changement de 0.03 environ et ça devient de plus en plus petit quand les nombres augmentent. Ainsi, le changement des poids sera compliqué si de nombreuses sigmoïdes sont présentes.

Dans le cas d'un réseau récurrent, les valeurs du temps présent ne passent pas beaucoup de sigmoïdes puisqu'elles traversent le réseau une seule fois. Cependant, les valeurs passées retraversent le réseau de nombreuses fois et vont donc être oubliées.

Remarque : Les fonctions d'activation ReLu ne sont pas affectées par ce problème.

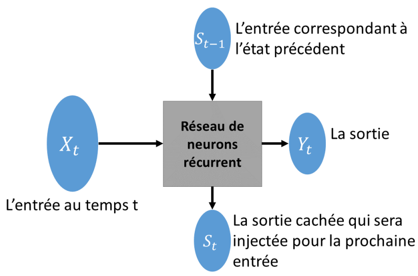


FIGURE 3.3 – Réseau de neurone récurent général

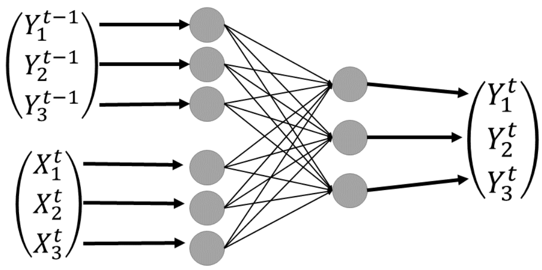


FIGURE 3.4 – Réseau de neurone récurrent standard

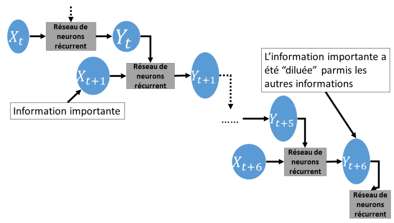


FIGURE 3.5 – Mémoire à court terme

3.2.2 LSTM

Un LSTM (Long Short Term Memory) est un réseau de neurones récurrent composé de cellules de LSTM de la forme suivante :

Ce réseau permet de se souvenir d'informations importantes correspondant à une mémoire courte (informations sur l'instant présent) mais sur une longue période.

Le probème de disparition de gradient n'affecte pas le LSTM car les fonctions d'activation utilisées ne sont pas des sigmoïdes (sauf pour l'apprentissage) mais des fonctions logisitiques ou des fonctions ReLu.

Ainsi, ce type de réseau convient parfaitement à notre objectif puisqu'il permet de prédire une valeur en tenant compte d'une séquence passée en conservant les éléments importants de ces séquences.

3.3 Les paramètres du modèle

Notre réseau a plusieurs paramètres (valeurs ayant une influence sur le modèle et qui sont modifiables sans que le programme rencontre des problèmes), qui sont :

1. La taille de la séquence : cette valeur peut être modifiée.
2. Le nombre d'itérations.
3. La taille du batch.
4. Le taux d'apprentissage.
5. Les fonctions d'activation.
6. « L'optimizer » et ses paramètres.
7. La fonction coût.
8. Les différentes couches et leur nombre de neurones.

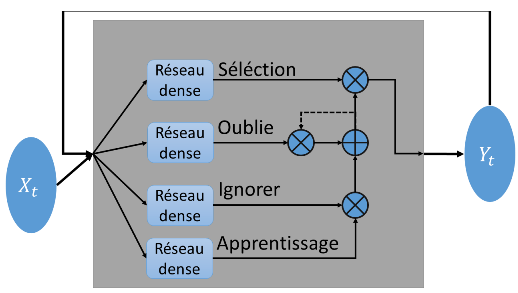


FIGURE 3.6 – LSTM

3.4 Les fonctions d'activations

Quelles fonctions peut-on utiliser avec Keras ? et qui sont pertinentes avec nos données ?

3.5 Les « optimizers »

Quels sont les optimizers que l'on peut utiliser ? Quels sont leurs paramètres ?

3.6 Les fonctions coûts

Quelles sont les fonctions coûts que l'on peut utiliser ?

3.7 L'architecture du réseau : les couches

Chapitre 4

Analyse et Test du réseau de neurones

Graphes, précisions, les différents tests effectués, le temps de calcul et analyse des résultats....

4.1 Modèle LSTM + Dense

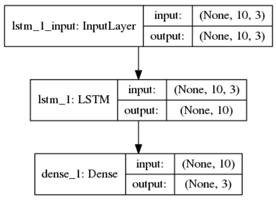


FIGURE 4.1 – Modèle LSTM + Dense

acc	val_loss	val_acc
0.9291	0.0183	0.9498

4.2 Modèle LSTM + Dense + Activation

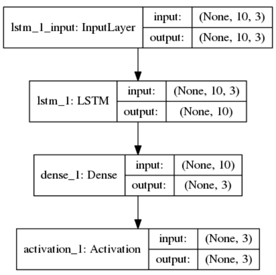


FIGURE 4.2 – Modèle LSTM + Dense + Activation

acc	val_loss	val_acc
0.9291	0.0307	0.9498

4.3 Modèle LSTM + Dropout + LSTM + Dropout + Dense + Activation

acc	val_loss	val_acc

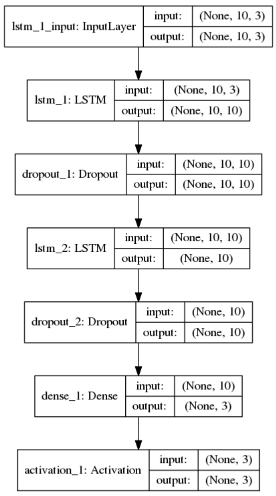


FIGURE 4.3 – LSTM + Dropout + LSTM + Dropout + Dense + Activation

Chapitre 5

Prédiction avec le réseau de neurones

Dans cette partie, nous allons expliquer comment prédire des données avec nos programmes et notre réseau de neurones.

5.1 Création de la prédiction

Pour création des prédictions, nous avons créé le programme « prediction.py » que vous trouverez en annexe. Ce programme permet charger le modèle de notre réseau de neurones et de lire un fichier dans lequel nous avons mis quelques notes (ticks, keys, velocity). Le programme normalise ensuite le fichier et le transforme en un tableau à 3 dimensions. Ensuite, le programme créé plusieurs plusieurs prédictions grâce à la fonction « model.predict(x) » de la bibliothèque Keras. Il y a un seul paramètre qui peut être changé : c'est le nombre de notes pour la chanson. Il faut tout de même mettre au moins le même nombre de notes que la taille de la séquence défini dans le modèle. La prédiciton fonctionne sur le même principe que le réseau de neurones, c'est-à-dire qu'il lit 10 notes et prédit la 11ème. Ensuite, il utilise les 10 notes après la première pour en prédire la 12ème etc...

Une fois que plusieurs notes ont été créées, le programme ajoute ces notes aux notes précédentes et les dé normalisent.

5.2 Lecture de la prédiction

Afin de pouvoir créer de la musique, le programme lit les notes normalisées qui ont été prédites et les utilisent pour créer de la musique.

Pour cela, nous avons fait un programme « creation_fichierMIDI » qui à partir d'un fichier contenant des valeurs de ticks, keys et velocity créé un fichier MIDI.

Conclusion

Les difficultés

Les bénéfices

Ouverture

Sources

Sources concernant les réseaux de neurones

1. « Réseaux de neurones : introduction et applications », vidéo présenté par Joseph Ghafari.
   `https://www.youtube.com/watch?v=KVNhk6uGmr8`
2. Site web relatant un projet sur la composition musicale par réseaux de neurones, par Daniel Johnson.
   `http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/`
3. « Apprentissage statistique » , écrit par Gerard Dreyfus et édité par Eyrolles (2008).
4. « Can a deep neural network compose music ? », article écrit par Justin Svegliato pour le blog « Medium.com ».
   `https://medium.com/towards-data-science/can-a-deep-neural-network-compose-music-f89b6ba4978d`
5. « Intelligence Artificielle », ensemble de vidéos présenté par Hugo Larochelle professeur à l'Université de Sherbrooke.
   `https://www.youtube.com/watch?v=stuU2TK3t0Q&list=PL6Xpj9I5qXYGhsvMWM53ZLfwUInzvYWsm`
6. LSTM
   `https://www.youtube.com/watch?v=WCUNPb-5EYI`

Sources concernant les bibliothèques Python (Keras)

1. Le modèle séquentiel `https://keras.io/getting-started/sequential-model-guide/`
2. Les fonctions d'activation `https://keras.io/activations/`
3. Les méthodes d'optimisation `https://keras.io/optimizers/`
4. Les fonctions de coût `https://keras.io/losses/`
5. Tuto pour utiliser Keras `https://machinelearningmastery.com/tutorial-first-neural-network-python-keras`

Sources concernant les fichiers MIDI

1. Spécifications MIDI : `https://web.archive.org/web/20120317213145/http://www.sonicspot.com/guide/midifiles.html`
2. Norme MIDI et les fichiers MIDI : `http://www.jchr.be/linux/format-midi.htm`
3. Standard MIDI-File Format Spec :
   `http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html`
4. The MIDI File Format : `https://www.csie.ntu.edu.tw/~r92092/ref/midi/`

Annexe A

Les fichiers MIDI

A.1 Les fichiers MIDI

Un fichier MIDI (Musical Instrument Digital Interface), contrairement au fichier audio, ne contient aucun son, à proprement parler, mais une série de « directives » que seul un instrument compatible MIDI peut comprendre. L'instrument MIDI d'après les « consignes » contenues dans le fichier MIDI peut alors produire le son.

Fréquences `http://newt.phys.unsw.edu.au/jw/notes.html`
Un fichier MIDI est composé d'un en-tête et d'un corps de fichier. L'en-tête décrit des informations nécessaires à la lecture du fichier.

Le corps du fichier est composé d'une ou plusieurs pistes.

Les fichiers MIDI sont des fichiers binaires : les données sont stockées en format binaire et non codées (texte, objets etc)

A.2 En-tête des fichiers MIDI

Structure de l'en-tête L'en-tête début toujours par les 4 octets x4D 54 68 64. Nous pouvons interpréter ces octets par la table ASCII où chaque octet est associé à un caractère de l'alphabet anglais. Nous trouvons 4D = M et de même pour les autres nombres. Finalement, l'en-tête débute toujours par les caractères MThd.

L'entête est de 14 octets (taille fixe) tel que récapitulé dans le tableau ci-dessous :

4 octets	x4D 54 68 64 (MThd)
4 octets	xN (Taille des données <D>)
N octets	<D> Données (exemple : format, piste, division)

TABLE A.1 – Structure de l'en-tête

Puisque l'en-tête est de taille fixe, N vaut toujours 14-(4+4) = 6 octets. La valeur dans le champs de 4 octets sera donc xN = x00 00 00 06.

L'en-tête décrit trois données : <format>, le format de fichier MIDI, <pistes> le nombre de pistes contenues dans le fichier, et <division> l'interval de temps.

On peut donc représenter l'en-tête de cette façon :

<table>
<tr><td>Chunk</td><td>Taille</td><td colspan="3">Données</td></tr>
<tr><td rowspan="2">4 octets</td><td rowspan="2">4 octets</td><td colspan="3">taille (= 6 octets)</td></tr>
<tr><td>2 octets</td><td>2 octets</td><td>2 octets</td></tr>
<tr><td></td><td></td><td><format></td><td><nbPistes></td><td><division></td></tr>
</table>

TABLE A.2 – Représentation de l'en-tête

Les formats MIDI Il existe trois formats de fichiers MIDI, décrits par les numéros 0, 1 et 2.

Le premier format (0) est le plus simple : le fichier ne présente qu'une seule piste. Le second format (1) décrit un fichier MIDI possédant une ou plusieurs piste pouvant être jouées simultanément.

Le troisième format (2) indique que le fichier MIDI possède une ou plusieurs pistes indépendantes.

→ Les fichiers MIDI que nous avons à interpréter sont présentés au format 1.

Les pistes MIDI Une piste MIDI est une suite d'événements tels que "La note est jouée" (événement MIDI). Ces pistes constituent donc la musique en elle-même. Cependant, les pistes peuvent aussi contenir des informations telles que le titre de la musique ou encore le copyright (Meta événement).

La division MIDI La division MIDI permet de définir la durée du delta-time, le delta time définissant l'attente entre la lecture de deux événements (entre deux notes par exemple ou la durée d'appuie d'une note).

	Un en-tête complet													
hex	4D	54	68	64	00	00	00	06	00	01	00	0B	01	E0
ASCII	M	T	h	d	Pas d'équivalent ASCII									
	Indicateur de début d'en-tête MIDI				Les données qui suivent sont stockées sur 6 octets				Fichier MIDI de format 1		11 pistes		division	

FIGURE A.1 – Exemple d'en-tête complet

Exemple d'en-tête La donnée "division" s'interprète différemment selon la valeur de son 15ème bit (c'est à dire le 1er bit en partant de la gauche). On appelle ce bit le MSB (Most Significant Bit). Dans l'exemple précédent ce bit vaut 0 : x01E0 = $\mathbf{0}000\ 0001\ 1110\ 0000_2$

Dans ce cas on a le tableau suviant :

Position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Hex	0				1				E				0			
Bin	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0
		000000111100000_2 = 480 delta-time par quart de ronde														

TABLE A.3 – Division cas MSB=0

Dans le cas om le MSB vaut 1 on a :

Bin	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0
		0000001_2=1 image par seconde							11100000_2 = 224 delta-time par image							

TABLE A.4 – Division cas MSB=1

→ Les fichiers MIDI que nous avons à interpréter sont présentés de la 1ère façon (15ème bit à 0). Nous avons besoin de ces informations pour interpréter correctement le rythme de la musique décrite dans le corps du fichier. En effet, un nombre de delta-time nous indiquera le temps entre deux événements.

A.3 Corps des fichiers

Représentation du corps Le corps du texte s'articule d'une façon similaire à l'en-tête : il commence toujours par "MTrk", qui est également suivi de 4 octets précisant la taille des données en question. Ces données, cependant, ne sont pas de même nature que dans l'en-tête : elles sont constituée d'une suite d'événements MIDI, SYSEX ou META ainsi qu'un nombre de delta-time associé

Piste		
Type	Taille	Données
4 octets	4 octets	N octets
MTrk	<N>	<delta-time> <événements>

TABLE A.5 – Représentation du corps de fichier

Le delta-time, équivalent au nombre de ticks, correspond au temps écoulé après la piste précédente avant de jouer/lire cette piste. Par exemple, la première piste contiendra souvent le nom de la musique et sera lue dès le départ afin d'afficher ce titre. Son nombre de ticks sera donc de 0.

Exemple de début de piste Dans cet exemple on représente le début de la piste (partie gauche du tableau), il s'agit d'un en-tête pour la piste. On représente ensuite un exemple de meta événement que l'on verra dans la partie suivante.

Début de la piste						Le meta événement			
MTrk	taille				Ticks	Type		taille	dépend du type
4D 54 72 6B	00	00	0B	F9	00	FF	03	<N>	<contenu>
	La taille totale de la piste est 3 065 octets				La piste commence 0 ticks après la précédente	FF03 = nom de la piste		la nom de la piste prendra N octets	N octets en ASCII pour donner le nom de la piste

TABLE A.6 – Exemple de piste

Cet exemple est bien sûr incomplet puisque la piste doit finir par un événement spécial qui marque la fin de piste.

Variable Length Quantity (VLQ) :

`https://en.wikipedia.org/wiki/VLQ`

Le but est de récupérer une donnée relative à une variable de plus de 1 octet, ne sachant qu'au fur et à mesure de la lecture le nombre de caractères sur laquelle cette donnée est codée. Pour cela, à chaque octet (cela correspond à deux caractères dans le fichier MIDI), par le premier bit de l'octet, on sait si l'octet suivant sera également relatif à la donnée en question. Ce premier bit est appelé le Most Significant Bit (MSB). Le premier bit de chaque octet n'est donc pas compris dans les données à récupérer. Si celui-ci est de 0, uniquement l'octet en question correspondra à la donnée à récupérer, si celui-ci est de 1, l'octet en cours (sauf le premier bit) et le suivant (sauf le premier bit), est réservée pour le codage de la variable et ainsi de suite.

A.4 Les événements MIDI

Meta événements Les métadonnées donnent des informations sur la partition.

Un meta événement débute toujours par l'octet xFF et est de la forme :

FF <type de meta événement> <taille des données> <données>

Par exemple, un meta événement de type 3 donne des informations sur le nom de la piste, c'est à dire que les données de ce meta événement seront interprétées comme une chaine de caractères (le nom de la piste).

Remarque :

La taille des données est codée en VLQ.

MIDI événement Un MIDI événement se compose de 2 parties : un type d'événement et les données. Le MIDI événement est toujours écrit sur 3 octets comme suit :

Type		Données			
1 octet		1 octet		1 octet	
H_1	H_2	H_3	H_4	H_5	H_6

TABLE A.7 – Représentation d'un MIDI événement

Les données s'interprètent différemment selon la valeur du type.

En effet, le type est composé de 2 nombres hexadécimaux H_1 et H_2. H_2 représente le canal sur lequel l'événement à lieu (le canal étant un instrument par exemple). H_1 quant à lui, donne l'événement qui à lieu (une note est appuyée, une note est relâchée, etc...).

On peut trouver une liste des événements principaux sur `https://www.csie.ntu.edu.tw/~r92092/ref/midi/midi_channel_voice.html` et `https://www.csie.ntu.edu.tw/~r92092/ref/midi/midi_channel_mode.html` .

Voici le MIDI événement correspondant à « Note Off » c'est à dire que la touche en question est relâchée :

Note off	Canal MIDI	Numéro de note	Volume
x8	x0-15	x0-127	x0-127

TABLE A.8 – Exemple Note off

Même chose pour « Note on » sauf que le codage hexadécimal est de 9.

Numéro de touches du piano Il y a 88 touches au total sur un piano (36 noires et 52 blanches). Chacune de ces touches correspondent à une note différentes. Il y a donc 88 notes au total sur un piano classique. On définit une numérotation des touches telle que les touches noires possèdent les numéros 1 à 36 et les touches blanches sont notées de 37 à 88. Le sens de numérotation se fait des graves vers les aiguës :

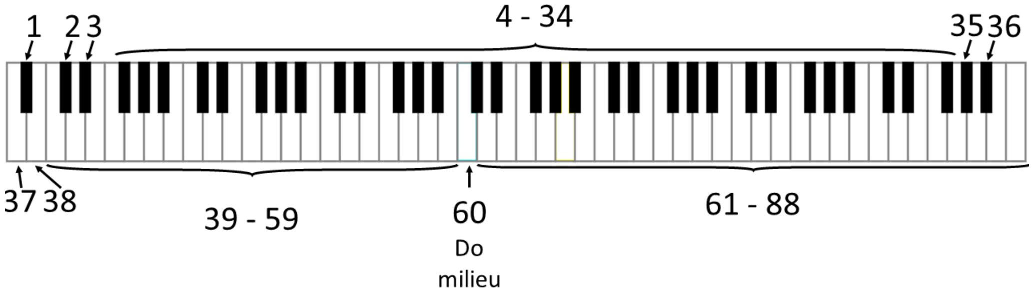


FIGURE A.2 – Numérotation des touches du piano

Exemple :

Par exemple, le message "Appuyer sur le do milieu (60ème touche) avec un volume moyen sur l'instrument 0 (instrument par défaut)" sera :

Type		**Données**			
1 octet		1 octet		1 octet	
"appuyer sur une touche"	"utiliser l'instrument 0"	Touche numéro x3C = 60 = "do milieu"		Volume	
9	0	3	C	4	0

TABLE A.9 – Exemple Note On

SysEx événement Ces événements permettent de faire appel à des appareils extérieurs et dépendent donc de l'appareil sur lequel le fichier MIDI est lu . Par exemple, si un synthétiseur spécial est connecté à l'ordinateur alors un événement sysex peut lui être envoyé (d'où SysEx = System Exclusiv = événement réservé pour ce système exclusivement). Cependant, si ce même fichier MIDI est lu sur un ordinateur ne possédant pas ce synthétiseur spécial alors l'événement ne sert à rien. Ainsi, ce genre d'événement n'est pas utilisé sur des fichiers dont le but est d'être partagé sur internet par exemple. En effet, puisqu'on ne sait pas quel appareil sera relié à l'ordinateur de l'utilisateur on ne peut pas utiliser ces événements. C'est pourquoi **nous ne les utiliserons pas et nous n'en tiendrons pas compte si nous en rencontrons un dans nos fichiers.**

Annexe B

Keras

B.1 Choix du langage de programmation

Choix du langage de programmation Nous avons choisi d'utiliser le langage Python pour coder le réseau de neurones. En effet, Python est un langage qui possède de nombreux outils afin de simplifier la mise en œuvre d'applications mathématiques. Aussi, TensorFlow (Google), Theano, MXNet et CNTK (Microsoft) sont quatre bibliothèques très utilisées pour mettre en place des réseaux de neurones bien qu'il en existe d'autres. Ces bibliothèques ne sont pas spécialisées dans la création de réseaux de neurones mais proposent un plus large éventail d'applications concernant l'apprentissage automatique (« machine learning »). Par exemple Theano permet de manipuler et d'évaluer des expressions matricielles en utilisant la syntaxe de NumPy, une autre bibliothèque Python qui permet la manipulation de matrices (similaire à Matlab).

De plus, les fonctions regroupées dans ces bibliothèques sont très optimisées et sont souvent compilées ce qui permet d'obtenir une vitesse d'exécution supérieure à l'utilisation du Python interprété. L'utilisation d'un GPU (Graphical Processing Unit / carte graphique) est aussi très facilitée grâce à ces programmes. Cela augmente aussi énormément la vitesse de calcul car ce type de processeur est spécialisé dans le calcul matriciel et parallèle contrairement au CPU (Central Processing Unit / processeur) qui est efficace en calcul séquentiel (un processeur possède une dizaine de cœurs logiques quand une carte graphique en possède plusieurs milliers).

Cependant, la non spécificité de ces bibliothèques peut conduire à des écritures lourdes pour construire un réseau de neurones alors que l'on n'utilise pas toutes les capacités offertes par la bibliothèque.

Ainsi, nous n'utiliserons pas directement ces bibliothèques mais implémenterons notre code à l'aide de Keras une bibliothèque Python qui agit comme une API (Application Program Interface) pour les bibliothèques précédemment citées. C'est-à-dire que Keras sert d'intermédiaire avec TensorFlow par exemple. Keras est une API de réseau de neurones de haut niveau : elle permet de programmer un réseau de neurone avec une syntaxe plus facilement appréhendable puisqu'elle est spécifiquement pensée pour ce type d'apprentissage automatique.

En utilisant Keras, un programme ne perd que très peu en vitesse d'exécution puisque ce sont les bibliothèques très optimisées qui vont être utilisées en arrière-plan.

Une seule bibliothèque à la fois peut être utilisée par Keras. Cependant, un programme écrit avec la syntaxe de Keras pourra être réutilisé après avoir changé de bibliothèque. Nous avons choisi d'utiliser Keras avec TensorFlow car étant tous deux développés par Google, leur couplage est facilité. En effet, TensorFlow a ajouté la prise en charge de Keras dans sa bibliothèque en 2017.

B.2 Syntaxe Keras

B.2.1 Création d'un modèle

Un réseau de neurones est considéré comme un « model ». On instancie un modèle séquentiel puis on peut ajouter des couches selon nos besoins. Une couche où chaque nœud est connecté avec les suivant (graphe complet) est appelée « Dense ». Pour les utiliser il faut tout d'abord les inclure dans le programme avec la commande « import ».

```
from keras.models import Sequential
from keras.layers import Dense
```

Pour créer une couche il faut donc instancier la classe Dense, son premier paramètre est le nombre de neurones et son deuxième le nombre de variables en entrées. Il est nécessaire de préciser le nombre de variable en entré de la première couche mais pas des suivantes : Keras le détermine directement.

On peut définir la dimension du vecteur en entrée avec input_shape ou input_dim et input_length.

```
# On ajoute une couche au modele "model"
# La couche possede 32 neurones et attend 784 variables en entré
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
# Est équivalent
model = Sequential()
model.add(Dense(32, input_dim=784))
```

On définit aussi les fonctions d'activation pour chaque couche. Keras propose de nombreuses fonctions d'activation comme la sigmoid ou ReLu. On peut définir la fonction d'activation directement lors de l'instanciation de la couche (attribut de Dense) ou après. Il faut d'abord importer le module « Activation ».

```
from keras.layers import Activation, Dense

# On ajoute une couche qui possède 64 neurones
# La fonction d'activation est une sigmoide

model.add(Dense(64))
model.add(Activation('sigmoid')
# Est équivalent
model.add(Dense(64, activation='sigmoid'))
```

Finalement, un modèle peut ressembler à :

```
from keras.models import Sequential
from keras.layers import Activation, Dense

# On instancie le modele
model = sequential()
# ajoute la premiere couche
model.add(Dense(n1, input_shape(300,), activation='relu'))
# on ajoute une deuxieme couche
model.add(Dense(n2, activation='sigmoid'))
# la derniere couche
model.add(Dense(n3, activation='tanh'))
```

Lorsque le modèle est terminé, il faut le compiler. C'est cette étape qui va faire appel à la bibliothèque en arrière-plan (TensorFlow dans notre cas).

B.2.2 Finalisation du modèle

Lorsque le modèle est terminé, il faut le compiler. C'est cette étape qui va faire appel à la bibliothèque en arrière-plan (TensorFlow dans notre cas).

La compilation va déterminer la meilleure façon de représenter le réseau de neurone afin de l'entrainer. Pour cela, il faut préciser trois arguments : la fonction de coût à minimiser, la méthode d'optimisation chargée de trouver cette minimisation et une métrique.

La fonction de coût est sous la forme d'une fonction python prenant en paramètre deux arguments : y_true et y_pred. Le premier est le vecteur des valeurs attendues et le second représente les prédictions. Des fonctions très utilisées comme l'erreur quadratique moyenne sont directement présentes dans Keras sous la forme d'alias (mse pour « Mean Squared Error » par exemple).

La méthode d'optimisation peut être implémentée par un alias présent dans Keras ou par une instance de la classe Optimizer. Les fonctions très utilisées comme la méthode du gradient sont donc présentes dans Keras. Une méthode de descente de gradient est implémentée par SGD (« Stochastique Gradient Descent ») c'est-à-dire l'algorithme du gradient stochastique.

Finalement une compilation de modèle se fait sous la forme suivante :

```
from keras import optimizers
```

```
from keras import losses

# Instanciation du modèle
# ...

# Compilation

# La fonction d'optimisation
# Ici on prend une méthode de descente de gradient
opti = optimizers.SGD()
# ou
opti = 'sgd'

# La fonction de coût
# Ici on prend l'erreur quadratique moyenne
cout = 'mse'
# ou
cout = losses.mean_squared_error
# ou
cout = 'mean_squared_error'

model.compile(optimizer=opti, loss=cout, metrics=['accuracy'])
```

B.2.3 Entraînement du réseau de neurones

Pour entrainer le modèle, on utilise la fonction fit. Elle prend plusieurs arguments : premièrement les données de test sous la forme d'un tableau utilisant la syntaxe de la bibliothèque Numpy. Ces données sont accompagnées de leur label en deuxième argument.

Le nombre d'itérations pour l'entrainement du modèle sur les données est géré par les attributs epochs et initial_epoch. Un « epoch » est défini comme une itération sur l'ensemble des données.

La mise à jour des poids peut être effectuée après avoir entrainé le modèle sur un certain nombre d'éléments (des musiques dans notre cas). Ce nombre peut être modifié par l'attribut batch_size (un « batch » étant un groupe d'éléments). Plus le nombre d'éléments pris en compte est grand, plus l'approximation sera bonne. Cependant, l'utilisation d'un nombre important d'éléments utilisera beaucoup plus de mémoire et allongera le temps de calcul. Un compromis doit donc être trouvé par l'expérience.

```
# Entrainement du modèle
# sur 10 itérations avec une mise à jour des poids à l'aide de 5 éléments
model.fit(data, labels, epochs=10, batch_size=5)
```

B.2.4 Prévision

La prévision est effectuée en exécutant la méthode predict sur le modèle. Les données sont données en paramètre sous la forme d'un tableau suivant la syntaxe Numpy.

```
# Calculer une prévision avec un vecteur X en entrée
previsions = model.predict(X)
```

B.2.5 Sauvegarder le modèle

Afin de ne pas devoir effectuer l'entrainement du réseau lors de chaque démarrage du programme, il est possible de sauvegarder un modèle dans un fichier. Il est aussi possible de reprendre l'entrainement au point où la dernière exécution s'était arrêtée.

Ainsi, l'architecture du réseau, ses poids, sa configuration (fonction de coût, d'optimisation) et son dernier état d'avancement son sauvegardés dans le fichier.

On utilise la fonction suivante :

```
from keras.models import load_model
```

```
# instanciation du modèle, compilation et entrainement
# ...

# Enregistrement du moèdle dans un fichier
model.save('fichier_du_modele.h5')

# Chargement du modèle stocké dans le fichier
model = load_model('fichier_du_modele.h5')
```

Annexe C

Les programmes

C.1 creation_donnees.py

```
import sys
import re
import subprocess
import os


import sys
import re
import subprocess
import os
import os.path

def creationDonnees(file):
                        #nom du fichier
                        pos = file.find('.mid')
                        nom = file[0:pos]
                        print("Traitement␣du␣fichier␣:␣",nom)
                        #creation du script MIDI complet
                        nomFichier = subprocess.call("mididump.py␣"+ file+"␣>␣"+
                            nom+"fMidiComplet.py", shell=True)
                        #creation du script MIDI simplifie
                        fichier = open(nom+"fMidiComplet.py", "r")
                        mon_fichier = open(nom+"fMidiSimple.py", "w")
                        mon_fichier.write("import␣midi\npattern=")
                        k=0
                        bad_words = ['ControlChangeEvent','PortEvent','
                           EndOfTrackEvent','SmpteOffsetEvent', 'TrackNameEvent'
                           , 'TextMetaEvent', 'SetTempoEvent','
                           CopyrightMetaEvent','TimeSignatureEvent','
                           KeySignatureEvent','ProgramChangeEvent','MarkerEvent'
                           ]
                        nombreTrack = 0
                        ligne=[]
                        for line in fichier :
                                 ligne.append(line)
                        fichier.close()
                        end = 0
                        nbTrack=0
                        for i in range(len(ligne)):
                                if "midi.Track(" in ligne[i]:
                                        nbTrack+=1
                                        if(i<len(ligne)-3) and end==0:
```

```
                                if "PortEvent" in ligne[i+1] and
                                     "midi.TrackNameEvent" in
                                    ligne[i+2] and  "midi.
                                    EndOfTrackEvent" in ligne[i
                                    +3]:
                                        end=nbTrack
                                else:
                                        if "TrackNameEvent" in
                                            ligne[i+1] and  "midi
                                            .EndOfTrackEvent" in
                                            ligne[i+2]:
                                                end=nbTrack
        print(end)
        fichier.close()
        fichier = open(nom+"fMidiComplet.py", "r")
        for line in fichier :
                clean = True
                if 'midi.Track(' in line :
                        k+=1
                for word in bad_words :
                    if word in line:
                        clean = False
                if clean == True:
                        if 'midi.Track(' in line:
                                if k==2:
                                        mon_fichier.
                                            write("[midi.
                                            Track(\\\n[")
                                if k==3:
                                        mon_fichier.
                                            write('␣␣␣
                                            midi.
                                            EndOfTrackEvent
                                            (tick=0,␣data
                                            =[])]),\n␣
                                            midi.Track
                                            (\\\n[␣')
                                if k==4:
                                        mon_fichier.
                                            write('␣␣␣
                                            midi.
                                            EndOfTrackEvent
                                            (tick=0,␣data
                                            =[])])])')
                                        print(k)
                        else:
                                if (k<4):
                                        mon_fichier.
                                            write(line)
        mon_fichier.write('\nmidi.write_midifile("creationMidi.
            mid",␣pattern)')
        mon_fichier.close()
        mon_fichier = open(nom+"fMidiSimple.py","r")
        k=−1
        os.chdir(os.path.dirname(os.getcwd()))
        os.chdir('Donnees')
        donnees = open(nom+".txt", "w")
        global nb
        nb+=1
        print("NB",nb)
        for line in mon_fichier :
```

```
                                if 'ControlChangeEvent' in line:
                                        k="0"
                                else:
                                        if 'NoteOnEvent' in line:
                                                k="0.5"
                                        else:
                                                if 'EndOfTrackEvent' in line:
                                                        k="1"
                                                else: k=-1
                                if k=="0.5":
                                        s = re.findall(r"[-+]?\d*\.\d+|\d+",
                                            line)
                                        #donnees.write(k)
                                        donnees.write(s[0])
                                        donnees.write("␣"+s[2])
                                        donnees.write("␣"+s[3])
                                        donnees.write("\n")
                        donnees.close()
                        mon_fichier.close()
                        os.chdir(os.path.dirname(os.getcwd()))
                        os.chdir('MIDI')
                        #os.remove(nom+"fMidiSimple.py")
                        #os.remove(nom+"fMidiComplet.py")

os.chdir('MIDI')
i=0
nb=0
for root, dirs, files in os.walk(os.getcwd()):
    for file in files:
        if file.endswith('.mid'):
            creationDonnees(file)
            i+=1
print(i)
```

C.2 analyse.py

```
import sys
import re
import subprocess
import os
import numpy as np
import statistics as stats
import sys
import re
import subprocess
import os
import os.path

max_ligne = 2000
min_ligne = 440
max_tick = 3800


def lecture(file):
        global nbLignesSup3500,nbLignesInf3500,nbLignesSupTICK,nbLignesInfTICK
        print("Lecture␣du␣fichier:␣", file)
        fichier = open(file, "r")
        nbLignes=0
        for line in fichier :
```

```
            s = re.findall(r"[-+]?\d*\.\d+|\d+", line)
            tick.append(float(s[0]))
            data1.append(float(s[1]))
            data2.append(float(s[2]))
            nbLignes+=1
        liste.append(nbLignes)
        if nbLignes>=max_ligne:
            nbLignesSup3500+=1
        if nbLignes<max_ligne and nbLignes>=min_ligne:
            nbLignesInf3500+=1
        if nbLignes>=max_ligne and float(s[0]) <=max_tick:
            nbLignesSupTICK+=1
        if nbLignes<max_ligne and float(s[0]) <= max_tick:
            nbLignesInfTICK+=1


nbLignesSup3500=0
nbLignesInf3500=0
nbLignesSupTICK=0
nbLignesInfTICK=0
nbTickSup=0
nbTickInf=0
event = []
tick = []
data1 = []
data2 = []
os.chdir(os.path.dirname(os.getcwd()))
os.chdir('Donnees')
liste=[]
for root, dirs, files in os.walk(os.getcwd()):
    for file in files:
        if file.endswith('.txt'):
            lecture(file)

for element in tick:
    if element > max_tick:
        nbTickSup+=1
    if element <=max_tick:
        nbTickInf+=1


print('\n')
print("Maximum␣lignes␣=␣",max(liste))
print("Minimim␣lignes␣=␣",min(liste))
print("Moyenne␣lignes␣=␣",int(np.mean(liste)))
print("Mediane␣lignes␣=␣",int(np.median(liste)))
print("Mediane␣groupe␣lignes␣=␣",stats.median_grouped(liste))
print("Nombre␣de␣lignes␣>␣1500␣=␣",nbLignesSup3500)
print("Nombre␣de␣lignes␣<␣1500␣=␣",nbLignesInf3500)
print("Nombre␣de␣lignes␣␣et␣tick␣>␣=␣",nbLignesSupTICK)
print("Nombre␣de␣lignes␣␣et␣tick␣<␣=␣",nbLignesInfTICK)
#print( list(sorted(set(liste))))


print('\n')
print("Maximum␣tick␣=␣",max(tick))
print("Minimim␣tick␣=␣",min(tick))
print("Moyenne␣tick␣=␣",np.mean(tick))
print("Mediane␣tick␣=␣",np.median(tick))
print("Ecart␣type␣tick␣=␣",np.std(tick))
print("Nb␣Sup␣tick␣=␣",nbTickSup)
```

```
print("Nb␣Inf␣tick␣=␣",nbTickInf)

print('\n')
print("Maximum␣data1␣=␣",max(data1))
print("Minimim␣data1␣=␣",min(data1))
print("Moyenne␣data1␣=␣",np.mean(data1))
print("Mediane␣data1␣=␣",np.median(data1))

print('\n')
print("Maximum␣data2␣=␣",max(data2))
print("Minimim␣data2␣=␣",min(data2))
print("Moyenne␣data2␣=␣",np.mean(data2))
print("Mediane␣data2␣=␣",np.median(data2))

#from collections import Counter
#print(Counter(liste))
```

C.3 main.py

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation,Dropout,LSTM,TimeDistributed,
    Bidirectional
from keras import losses
from keras import optimizers
import numpy as np
from keras import regularizers
from keras.utils import plot_model
from keras.models import load_model
import matplotlib.pyplot as plt
from keras.callbacks import EarlyStopping,ModelCheckpoint
import sys
import re
import subprocess
import os

def transformerArrayEn2D(nomFichier,dim):
        donnees = open(nomFichier,"r")
        lines = donnees.readlines()
        i=0
        data = [[0 for y in range(3)] for x in range(dim)]
        for line in lines:
                s = re.findall(r"[-+]?\d*\.\d+|\d+\t\n\r\f\v", line)
                if i<=dim:
                        data[i][0]=float(s[0])
                        data[i][1]=float(s[1])
                        data[i][2]=float(s[2])
                i+=1
        donnees.close()
        return data

def transformerArrayEn3D(nomFichier,dim1,dim2,dim3):
        data = transformerArrayEn2D(nomFichier,dim1*dim2)
        data = np.reshape(data,(dim1, dim2, dim3))
        X = data[np.mod(np.arange(data.shape[0]),dim2)].reshape(dim1,dim2,dim3)
        return X

def creationDonneesApprentissage(nomFichier,dim1,dim2,dim3,nbE,ePc,tS):
        X = transformerArrayEn3D(nomFichier, dim1,dim2, dim3)
```

```
        x = np.zeros(shape=(nbE, tS, dim3))
        y = np.zeros(shape=(nbE, dim3))
        print("X shape = ",X.shape) #(272, 1500, 3)
        print("x shape = ",x.shape) #(405280, 10,3)
        print("Y shape = ",y.shape) #(405280, 3)
        print(enumerate(X))
        #n va de 0 a 271
        for n, X in enumerate(X):
                for i in range(ePc):
                        x[i+n*ePc][:][:] = X[i:(i+tS), :] #ePc = 1490 #i+nePc =
                            0 a 405279
                        y[i+n*ePc][:] = X[i+tS, :] #i+tS va de 0 a 1499
        print("x shape = ",x.shape) #(405280, 10,3)
        print("Y shape = ",y.shape) #(405280, 3)
        return x,y


def creationDonneesPrediction(nomFichier,dim1,dim2,dim3):
        X = transformerArrayEn3D(nomFichier, dim1,dim2, dim3)
        x = np.zeros(shape=(nb_echantillon, taille_sequence, note_dim))
        y = np.zeros(shape=(nb_echantillon, note_dim))
        for n, X in enumerate(X):
                for i in range(echantillons_par_chanson):
                        x[i+n*echantillons_par_chanson][:][:] = X[i:(i+
                            taille_sequence), :]
        return x



taille_sequence = 10
note_dim = 3
nb_chanson = 241
nbNotes_par_chanson = 2000
echantillons_par_chanson = nbNotes_par_chanson - taille_sequence
nb_echantillon = nb_chanson*echantillons_par_chanson
batch_size = 15
epochs = 10
taux_apprentissage = 0.01
#opt = optimizers.rmsprop(taux_apprentissage)
decay_rate = taux_apprentissage / epochs
momentum = 0.8
opt = optimizers.SGD(lr=taux_apprentissage, momentum=momentum, decay=decay_rate,
    nesterov=False)
#opt = optimizers.Adadelta(lr=taux_apprentissage, epsilon=1e-6)
#cout = 'categorical_crossentropy'
cout = 'mean_squared_error'
nomFichierDuModele = 'modele.h5'
imageDuModele = 'modele.png'
nomFichierDesPoids = 'poids.h5'



x,y = creationDonneesApprentissage("donneesNormalisees.txt",nb_chanson,
   nbNotes_par_chanson, note_dim,nb_echantillon,echantillons_par_chanson,
   taille_sequence)
print(x.shape)
nb_chanson_test  = 94
nbNotes_par_chanson_test  = 440
echantillons_par_chanson_test  = nbNotes_par_chanson_test  - taille_sequence
nb_echantillon_test  = nb_chanson_test *echantillons_par_chanson_test
```

```
x_test,y_test = creationDonneesApprentissage("donneesTest.txt",nb_chanson_test,
    nbNotes_par_chanson_test, note_dim,nb_echantillon_test,
    echantillons_par_chanson_test,taille_sequence)

model = Sequential()
model.add(LSTM(taille_sequence, input_shape=(taille_sequence, note_dim),
    return_sequences=True))
#model.add(TimeDistributed(Dense(taille_sequence, activation='sigmoid')))
model.add(Dropout(0.2))
model.add(LSTM(1000))
model.add(Dense(128, activation='sigmoid'))
model.add(Dense(y.shape[1], activation='sigmoid'))
plot_model(model, to_file=imageDuModele, show_shapes=True, show_layer_names=True
    )
model.summary()



callbacks = [
    EarlyStopping(monitor='val_loss', patience=2, verbose=0),
    ModelCheckpoint("weights.{epoch:02d}-{val_loss:.2f}.hdf5",monitor='val_loss'
        , save_best_only=True, verbose=0),
]


model.compile(loss=cout, optimizer=opt,metrics=['accuracy'])

history = model.fit(x,y,verbose=1, validation_data=(x_test, y_test),batch_size=
    batch_size, epochs=epochs,callbacks=callbacks, shuffle=False)


#courbe de la precision sur les ensembles de donnees d'apprentissage et de
    validation au cours des iterations d'apprentissage.
#plt.plot(history.history['acc'])
#plt.plot(history.history['val_acc'])
#plt.title('Precision du modele')
#plt.ylabel('Precision')
#plt.xlabel('Iterations')
#plt.legend(['Apprentissage', 'Test'], loc='upper left')
#plt.show()
## courbe de la perte/cout sur les ensembles de donnees d'apprentissage et de
    validation au cours des iterations d'apprentissage.
#plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
#plt.title('Cout du modele')
#plt.ylabel('Cout')
#plt.xlabel('Iterations')
#plt.legend(['Apprentissage', 'Test'], loc='upper left')
#plt.show()

#obtenir les valeurs des poids par couche (utiliser le logicie HDFView)
model.save_weights(nomFichierDesPoids)

#EVALUATION
#loss_and_metrics = model.evaluate(x_test, y_test, batch_size=1)
#print ("Loss et metrics ",loss_and_metrics)
model.save(nomFichierDuModele)
```

C.4 prediction.py

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation,Dropout,LSTM,TimeDistributed
from keras import losses
from keras import optimizers
import numpy as np
from keras.utils import plot_model
from keras.models import load_model
import matplotlib.pyplot as plt
from keras.callbacks import EarlyStopping,ModelCheckpoint
import sys
import re
import subprocess
import os
from decimal import Decimal

def transformerArrayEn2D(nomFichier,dim):
        donnees = open(nomFichier,"r")
        lines = donnees.readlines()
        i=0
        data = [[0 for y in range(3)] for x in range(dim)]
        for line in lines:
                s = re.findall(r"[-+]?\d*\.\d+|\d+\t\n\r\f\v", line)
                if i<dim:
                        data[i][0]=float(s[0])
                        data[i][1]=float(s[1])
                        data[i][2]=float(s[2])
                i+=1
        donnees.close()
        return data

def transformerArrayEn3D(nomFichier,dim1,dim2,dim3):
        data = transformerArrayEn2D(nomFichier,dim1*dim2)
        data = np.reshape(data,(dim1, dim2, dim3))
        X = data[np.mod(np.arange(data.shape[0]),dim2)].reshape(dim1,dim2,dim3)
        return X

def creationDonneesPrediction(nomFichier,dim1,dim2,dim3,nbE,ePc,tS):
        X = transformerArrayEn3D(nomFichier, dim1,dim2, dim3)
        x = np.zeros(shape=(nbE, tS, dim3))
        y = np.zeros(shape=(nbE, dim3))
        for n, X in enumerate(X):
                for i in range(ePc):
                        x[i+n*ePc][:][:] = X[i:(i+tS), :]

        return x

def normalisation(fileRead,fileWrite):
    lines = fileRead.readlines()
    tick = []
    data1 = []
    data2 = []
    for line in lines:
            s = re.findall(r"[-+]?\d*\.\d+|\d+", line)
            tick.append(float(s[0]))
            data1.append(float(s[1]))
            data2.append(float(s[2]))
    fileRead.close()
```

```
    max_tick = 3500
    min_tick = 0
    max_data1 = 127
    min_data1 = 0
    max_data2 = max_data1
    min_data2 = min_data1
    for i in range(0,len(tick)):
            t = Decimal((float(tick[i])-min_tick)/(max_tick- min_tick))
            if (t==0):
                t=0.0
            if (t==1):
                t=1.0
            fileWrite.write(str(t)+" ")
            d1 = Decimal((float(data1[i])-min_data1)/(max_data1- min_data1))
            if (d1==0):
                d1=0.0
            if (d1==1):
                d1=1.0
            fileWrite.write(str(d1)+" ")
            d2 = Decimal((float(data2[i])-min_data2)/(max_data2 - min_data2))
            if (d2==0):
                d2=0.0
            if (d2==1):
                d2=1.0
            fileWrite.write(str(d2)+"\n")
            if (t or d1 or d2)>1.0 or (t or d1 or d2)<0.0:
                print("Probleme de normalisation!")
                print(i)
    fileWrite.close()



model = load_model('modele.h5')
nbNotesAPredire = 9
taille_sequence = 10
note_dim = 3
nb_chanson = 1
nbNotes_par_chanson = 20
echantillons_par_chanson = nbNotes_par_chanson - taille_sequence
nb_echantillon = nb_chanson*echantillons_par_chanson

#Normalisation du fichier d'entree et creation d'un fichier pour les 10 notes
nomTestPasNormalise = "testD.txt"
fichierPasNormalise = open(nomTestPasNormalise,"r")
nomTestNormalise = "testN.txt"
fichierNormalise = open(nomTestNormalise,"w")
normalisation(fichierPasNormalise,fichierNormalise)
fichierNormalise.close()
fichierPasNormalise.close()

#creation du fichier final
nomFichierFinal = "testFinal.txt"
fichierFinal = open(nomFichierFinal,"a")
fichierFinal.write(open(nomTestNormalise).read())
fichierFinal.close()
x = creationDonneesPrediction(nomFichierFinal,nb_chanson, nbNotes_par_chanson,
   note_dim,nb_echantillon,echantillons_par_chanson,taille_sequence)
prediction = model.predict(x, verbose=0, batch_size=1)
with open(nomFichierFinal, "a") as fic:
    for item in prediction:
```

```
        print(item)
        fic.write(str(item[0])+"␣"+str(item[1])+"␣"+str(item[2])+"\n")

fichierFinal.close()
subprocess.call("python3␣denormalisation.py␣"+nomFichierFinal+"␣>␣
   predictionFinale.txt", shell=True)
subprocess.call("python3␣creation_midi.py␣predictionFinale.txt", shell=True)
os.remove(nomFichierFinal)
os.remove("predictionFinale.txt")
os.remove(nomTestNormalise)
os.remove("donneesDenormalises.txt")
subprocess.call("timidity␣newMusic.mid", shell=True)
```