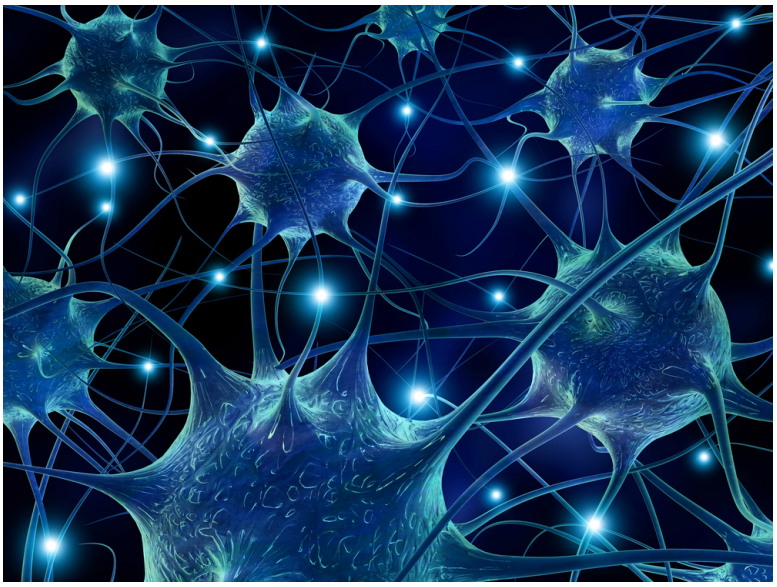
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE



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.

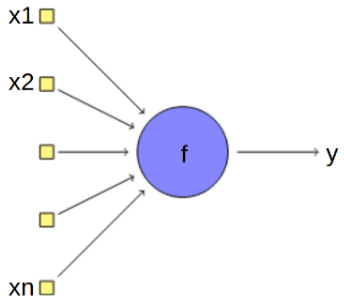


FIGURE 1.1 – Modélisation d'un neurone

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.

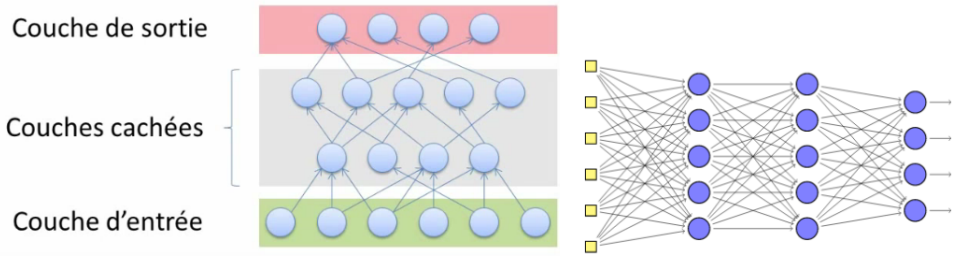


FIGURE 1.2 – Modèle de perceptron multicouche

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 :

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

avec b le seuil d'activation ou biais

Il s'agit de la fonction de Heaviside définie par $f(x_1, .., x_p) = H(\sum_{i \in I} w_i x_i - w_0)$ 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 :

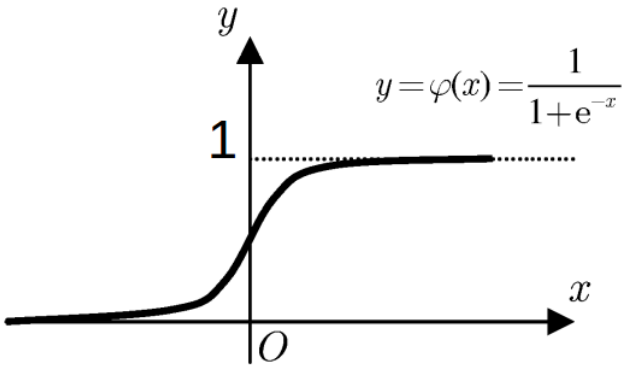


FIGURE 1.3 – Modèle de perceptron multicouche

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 :

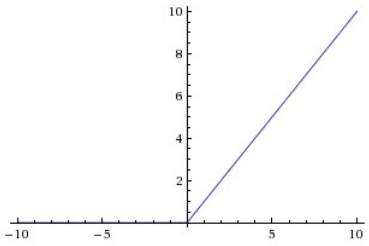


FIGURE 1.4 – Modèle de perceptron multicouche

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 :

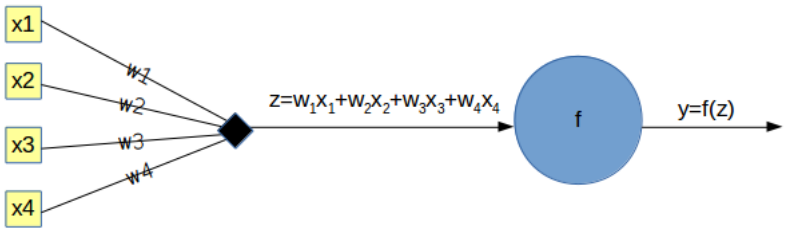


FIGURE 1.6 – Schéma d'un neurone avec 4 entrées

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 :

$$f_i(w) \simeq \overline{y_i} \Leftrightarrow \min_w \left(\sum_{i\in I} (f_i(w) - \overline{y_i})^2\right)$$

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 :

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

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 $\Pi(x_k, w) = \left(f(\nu_k) - \overline{y_k}\right)^2 = \left(y_k - \overline{y_k}\right)^2$ 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).

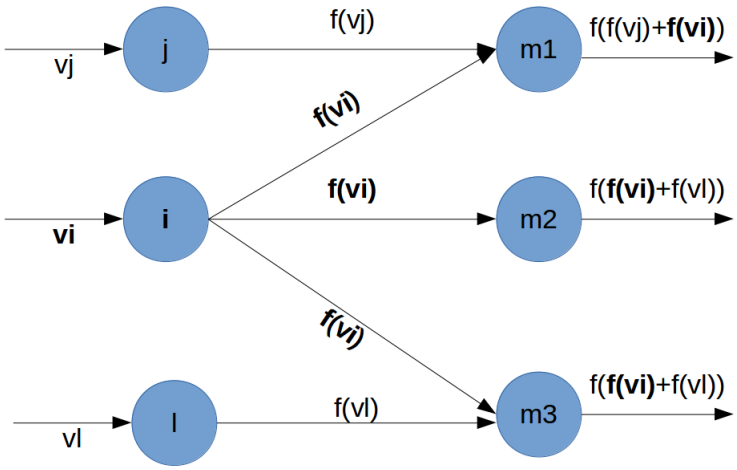


FIGURE 1.7 – Schéma des neurones m et i (sans les poids)

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

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

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.

$J_k = \frac{1}{2} \sum_{i=1}^{n_L} (y_i - \overline{y_i})^2_{x=x_k}$: 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**
        $\nu_j^l \leftarrow \sum_{i=1}^{n_{l-1}} w_{ij}^{l-1} a_i^{l-1}$
        $a_j^l \leftarrow f(\nu_j^l)$
      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**
        $\delta_j^l \leftarrow f'(\nu_j^l) \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^l$
      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
    $J_k \leftarrow \frac{1}{2} \sum_{i=1}^{n_L} \left(a_i^L - \overline{y_i}\right)^2$
  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 :

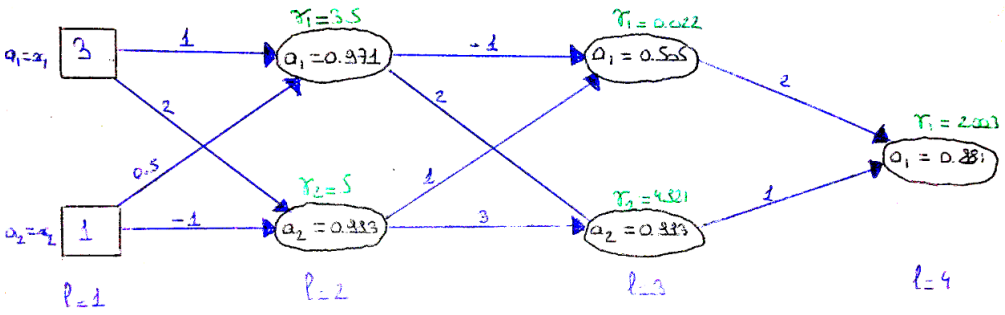


FIGURE 1.8 – Exemple propagation avant

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 :

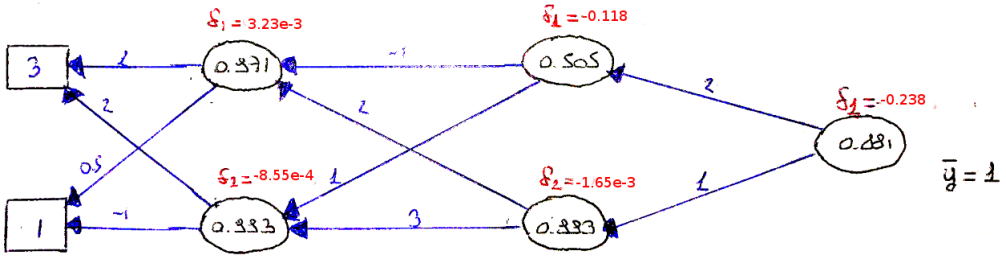


FIGURE 1.9 – Exemple propagation arrière

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 :

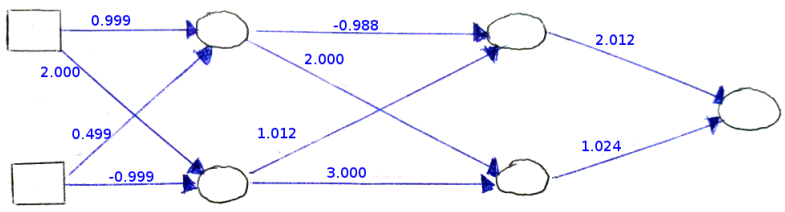


FIGURE 1.10 – Exemple modification des poids

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.

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)

2.2 Les données

Nos données seront des fichiers contenant des informations extraites de plusieurs fichiers MIDI. Nous allons avoir un ensemble de données que l'on partagera en deux : les données pour l'apprentissage et les données pour les tests.

2.2.1 Les fichiers

Où a-t-on trouvé les données ? Sous quelles formes sont-elles ? Combien d'exemples pour l'apprentissage et pour les tests ?

2.2.2 Extraction d'informations des données

2.2.3 Analyse de nos données

corrélation...

Chapitre 3

Composition musicale : le réseau de neurones

Nous allons définir dans cette partie l'architecture de notre réseau.

3.1 Les différentes couches

La couche d'entrée et de sortie Si une note est représentée par 3 valeurs alors une note sera représentée par 3 neurones de la couche d'entrée. La couché d'entrée contiendra plusieurs notes. Si la couche d'entrée contient 5 notes alors il y aura $3 \times 5 = 15$ neurones. La couche de sortie contiendra 3 neurones correspondant à la note qui devra être jouée après la séquence de notes de la couche d'entrée.

Les couches cachées Le nombre de couches cachées est arbitraire et il dépendra de nos résultats.

3.2 L'architecture du réseau

Fonction d'activation, formule des changements de poids avec le pas d'apprentissage, réseau récurrent, profond ? etc...

Chapitre 4

Programmation du réseau de neurones

Dans cette partie, nous allons expliquer comment nous avons programmer notre réseau de neurones.

4.1 Keras

https://stackoverflow.com/questions/44747343/keras-input-explanation-input-shape-units-batch-size-di
https://stackoverflow.com/questions/47988983/how-does-keras-read-input-data

4.2 Création du modèle

A utiliser :

1. Conv2D $model.add(Conv2D(32,(3,3),activation =' relu', input_shape = (100,100,3)))$
2. Dense $model.add(Dense(64, activation =' relu', input_dim = 20))$
3. LSTM $model.add(LSTM(32, returnequences = True, stateful = True, batch_input_shape = (batch_size, timesteps, dat$

Modèle faux...juste pour donner une idée

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation,Dropout
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
#LES DONNEES
#si  on a un fichier particulier
x_train = np.loadtxt("x_train.csv", delimiter=",")
y_train = np.loadtxt("y_train.csv", delimiter=",")
x_test = np.loadtxt("x_test.csv", delimiter=",")
y_test = np.loadtxt("y_test.csv", delimiter=",")
#Y = dataset[:,3]
#np.savetxt("X.csv",X, delimiter=",")
#np.savetxt("Y.csv",Y, delimiter=",")
#seed = 7
#np.random.seed(seed)
#tableau de random de taille 10 x 1
#x_train = np.random.random((9, 1))
#y_train = keras.utils.to_categorical(np.random.randint(9, size=(9, 1)),
    num_classes=9)
#tableau de random de taille 20 x 1
#x_test = np.random.random((20, 1))
#y_test = keras.utils.to_categorical(np.random.randint(9, size=(20, 1)),
    num_classes=9)
```

```
#np.savetxt('x_train.txt', x_train)
#np.savetxt('y_train.txt', y_train)
#np.savetxt('x_test.txt', x_test)
#np.savetxt('y_test.txt', y_test)
nomFichierDuModele = 'modele.h5'
nomFichierDesPoids = 'poids.h5'

#CREATION DU MODELE

#creation d'un reseau de neurones Perceptron multi-couches vide
model = Sequential()
#ajout d'une couche completement connectee avec 32 neurones et cette couche
    attend 100 valeurs en entree avec la fonction d'activation relu
#input_dim = nombre de colonnes dans nos donnees
model.add(Dense(20, activation='relu', input_dim=3,name='couche1'))
#apres la premiere couche il n'est pas necessaire de specifier imput_dim
model.add(Dense(20, activation='relu',name='couche2'))
model.add(Dense(20, activation='relu',name='couche3'))
model.add(Dense(10, activation='relu',name='couche4'))
model.add(Dense(1, activation='relu', name='couche5'))
#visualiser  le modele du reseau
plot_model(model, to_file='modele.png', show_shapes=True, show_layer_names=True)
#visualisation sous forme de tableau de l'architecture du reseau
model.summary()


#METHODE DOPTIMISATION ET COMPILATION DU MODELE

#le taux d'apprentissage
learning_rate = 0.01
#definition de la methode d'optimisation: ici on prend descente de gradient
    stochastique SGD
sgd = optimizers.SGD(learning_rate)
#definition du cout: erreur quadratique moyenne
cout = 'mean_squared_error'
#une metric= fonction qui permet de juger la performance du modele
model.compile(loss=cout,optimizer=sgd,metrics=['accuracy'])



#APPRENTISSAGE DU MODELE
#epochs = nombre d'iterations
#batch_size = nombre d'exemples par mise a jour du gradient
#validation_split = pourcentage de donnees utilisee pour les tests
#1 epoch = 1 propagation avant et arriere pour tous les exemples d'apprentissage
#batch size =nombre d'exemples d'apprentissage dans une seule propagation avant/
    arriere. Plus ce chiffre est grand, plus on aura besoin de memoire.
#nombre d'iteration = nombre de propagations, chaque propagation (avant+arriere
    =1 propagation) utilise [batch size] nombres d'exemples d'apprentissage.
#Exemple:  1000 exemples d'apprentissage et batch size=500 alors il y a 2
    iterations pour faire 1 epoch..
#early_stopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=2,
    verbose = 0)
history = model.fit(x_train, y_train,epochs=100,batch_size=1, verbose=1,shuffle=
    True ) #, callbacks=[early_stopping])


#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

#PREDICTION
previsions = model.predict(np.array([[-0.0294117647     ,-0.0002883506,
    0.0092300473]]))
np.savetxt('prediction.txt', previsions)

#sauvegarder le modele
model.save(nomFichierDuModele)




# Chargement du modele stocke dans le fichier pour le reutiliser
model = load_model(nomFichierDuModele)
```

4.3 Normalisation de nos données

4.4 Apprentissage

4.5 Interprétation de la sortie

4.5.1 Dé-normalisation

4.5.2 Création d'un fichier MIDI

4.5.2.1 1er exemple

```
import midi

# Instantiate a MIDI Pattern (contains a list of tracks)
pattern = midi.Pattern()
# Instantiate a MIDI Track (contains a list of MIDI events)
track = midi.Track()
# Append the track to the pattern
pattern.append(track)
# Instantiate a MIDI note on event, append it to the track
```

```
on = midi.NoteOnEvent(tick=0, velocity=20, pitch=midi.G_3)
track.append(on)
# Instantiate a MIDI note off event, append it to the track
off = midi.NoteOffEvent(tick=100, pitch=midi.G_3)
track.append(off)
# Add the end of track event, append it to the track
eot = midi.EndOfTrackEvent(tick=1)
track.append(eot)
# Print out the pattern
print pattern
# Save the pattern to disk
midi.write_midifile("example.mid", pattern)
```

4.5.2.2 2eme exemple

```
from midiutil.MidiFile import MIDIFile
# create your MIDI object
mf = MIDIFile(1)     # only 1 track
track = 0   # the only track
time = 0    # start at the beginning
mf.addTrackName(track, time, "Sample␣Track")
mf.addTempo(track, time, 120)
# add some notes
channel = 0
volume = 100
pitch = 60           # C4 (middle C)
time = 0             # start on beat 0
duration = 1         # 1 beat long
mf.addNote(track, channel, pitch, time, duration, volume)

pitch = 64           # E4
time = 2             # start on beat 2
duration = 1         # 1 beat long
mf.addNote(track, channel, pitch, time, duration, volume)

pitch = 67           # G4
time = 4             # start on beat 4
duration = 1         # 1 beat long
mf.addNote(track, channel, pitch, time, duration, volume)



mf.addNote(track, channel, 44, 3, 1, volume)

mf.addNote(track, channel, 50, 6, 1, volume)
# write it to disk
with open("essais.mid", 'wb') as outf:
    mf.writeFile(outf)
```

Chapitre 5

Le plan d'expérience

Chapitre 6

Les résultats

Statistique, optimisation, corrélation, temps de calcul, taux d'apprentissage, minimisation directionnelle = line search,gradient stochastique

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

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 :

Chunk	Taille	Données		
4 octets	4 octets	taille (= 6 octets)		
		2 octets	2 octets	2 octets
		<format>	<nbPistes>	<division>

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).

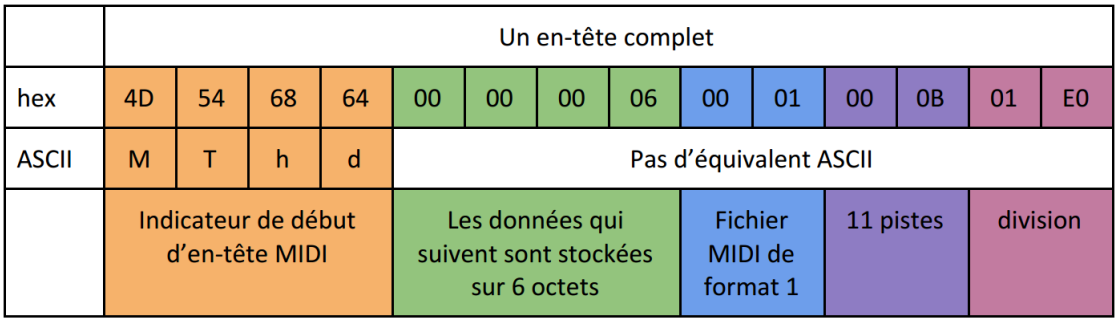


FIGURE A.1 – Exemple d'en-tête complet

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 = **0**000 0001 1110 0000_2

Dans ce cas on a le tableau suivant :

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
		0000000111100000_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 :

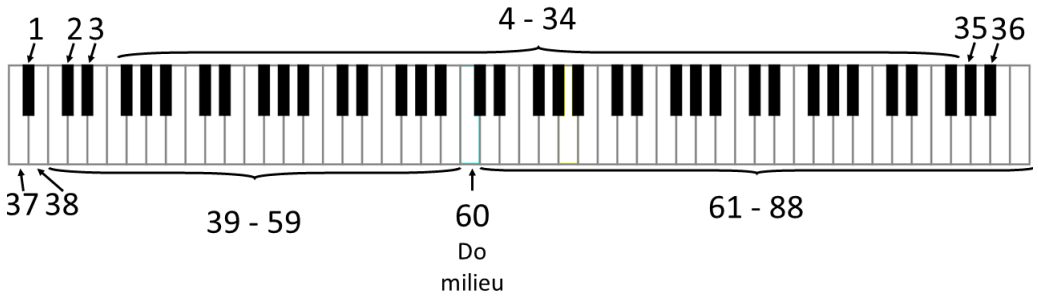


FIGURE A.2 – Numérotation des touches du piano

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')
```