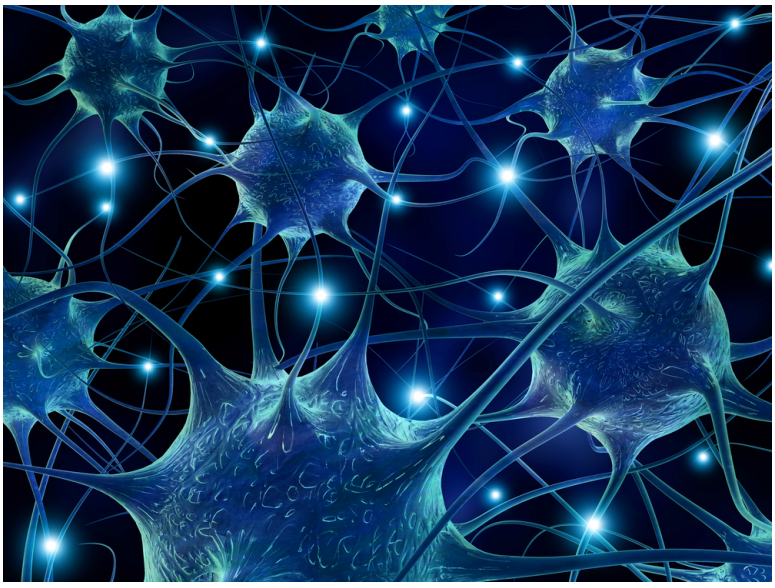
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE



Projet de Recherche Opérationnelle

Composition musical par réseau de neurones

DRIGUEZ CLAIRE
CATELAIN Jeremy
RAMAGE Lucas
GM4

Tuteur : M. Knippel

Octobre - Décembre 2017

Table des matières

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.

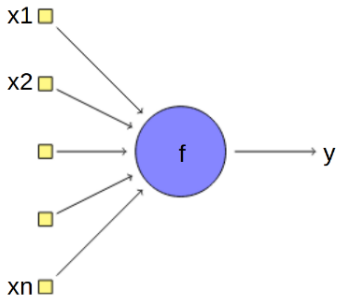


FIGURE 1.1 – Modélisation d'un neurone

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.

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.

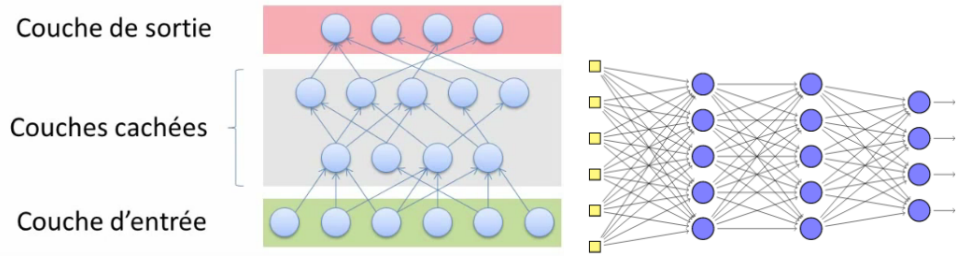


FIGURE 1.2 – Modèle de perceptron multicouche

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. Toutes les flèches 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 introduit, celui-ci correspond à un neurone d'une couche cachée. De plus, toutes les flèches d'un nœud ont donc le même poids car chaque nœud a une valeur de sortie unique. Par ailleurs, le comportement du réseau neuronal est déterminé par l'ensemble des poids w_i et des biais ou seuil d'activation w_0 propre à chaque nœud, donc il faut les ajuster à une valeur correcte. Cela est réalisable lors de la phase d'apprentissage. Il faut aussi définir la qualité de chaque sortie donnée (bien ou pas bien) 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 défini 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 :

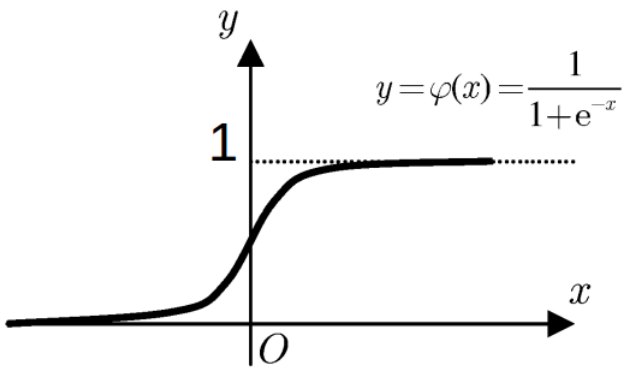


FIGURE 1.3 – Modèle de perceptron multicouche

Celle-ci possède des propriétés intéressantes. Celle-ci est continue et dérivable à l'infini. 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. Par contre, celle-ci peut être difficilement programmable car le calcul d'exponentiel négative correspond à des nombres très proche de 0 et donc inférieur à l'epsilon machine. $1 + e^{-x}$ peut parfois être équivalent à 1. Un codage particulier des nombres tel que la normalisation est alors à effectuer.

1.2 Apprentissage

1.2.1 L'apprentissage

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

Définition L 'apprentissage est une phase du développement d'un réseau de neurones 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 1.4 – 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 :

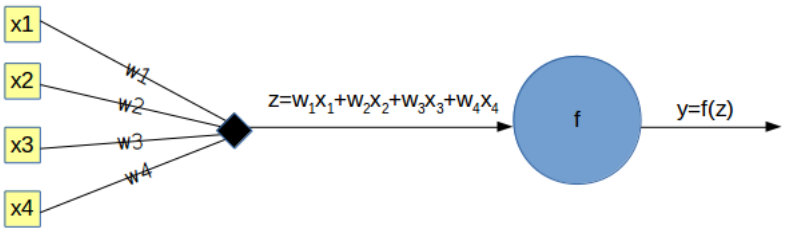


FIGURE 1.5 – Schéma d'un neurone avec 4 entrées

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 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 à propagation avant

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.

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 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 :

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

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. 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).

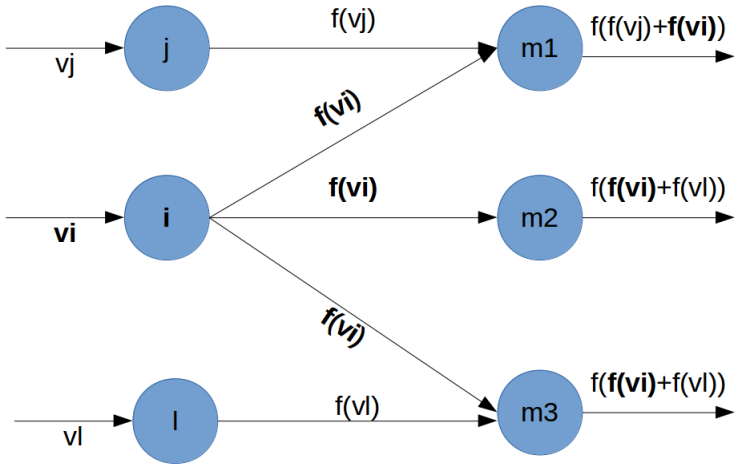


FIGURE 1.6 – Schéma des neurones m et i (sans les poids)

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

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 de 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.
3. Calcul 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.

1.2.2.3 Modification des paramètres (poids)

Chapitre 2

Réseau de neurones et composition musicale

Résumé du site

`https://medium.com/towards-data-science/can-a-deep-neural-network-compose-music-f89b6ba4978d` Le projet peut se faire en 2 étapes. La première étape est la création d'un modèle, où tous les paramètres d'un réseau de neurone généraux à toute partition de musique sont alors déterminés et fixés. La deuxième étape est la prédiction de l'objet à déterminer, ici une note de musique. En effet, une fois que le modèle est finalisé, le réseaux de neurone est alors opérationnel, et celui-ci peut être utilisé pour prédire la note suivante de la partition attendue.

La première étape est la plus complexe. On peut diviser cette étape en la détermination de l'entrée puis celle du mécanisme du réseau de neurone et enfin la sortie.

Il faut tout d'abord récupérer un échantillon de données, ces données peut être en format MP3, appelé format MIDI. Il faut alors trouver un moyen de traduire les informations données par cet échantillon en matrice contenant les informations nécessaires. Pour cela, il faut déterminer quelles informations doivent être prises en compte pour réaliser cette prédiction.

Prenons l'exemple du piano. On peut alors tout d'abord considérer les attributs liées à la partition : la pulsation choisie, le nombre de mesures et la clé du morceau en question. On peut par exemple fixer un nombre de pulsation ou un nombre de mesures pour chaque morceau à effectuer.

Ensuite, on peut considérer les attributs liés à une note de musique, par exemple l'attribut appelé "Position". En effet, comme un piano est composé de 88 touches, à chaque attribut "Position" peut être associé un nombre entier entre 1 et 88, se référant à la touche utilisée. Ensuite, le rythme associé à chaque note peut être un attribut, par exemple déterminer si on est dans le cas d'une ronde, d'une blanche, d'une croche etc. Par ailleurs, la spécificité d'un morceau réside dans l'enchaînement des notes. Il est donc important de connaître les informations sur la note précédant et succédant la note considérée. De plus, la place de la note dans une mesure peut être également à déterminer.

Enfin, la sortie peut associer aux 88 touches du piano, son état prédictif sous forme de probabilité déduisant ainsi, en considérant le maximum de probabilité, la note à jouer.

Ensuite, il faut déterminer quel type de réseaux de neurone utilisé, si on est dans le cas d'un réseaux de neurone acyclique ou cyclique ainsi que les opérations intrinsèque au mécanisme.

Par ailleurs, une fois que le modèle est construit, il faut le tester. Le test consiste à comparer le résultat de l'algorithme avec le véritable résultat à obtenir, en utilisant un échantillon spécifique et adaptant, les paramètres jusqu'à ce que l'erreur soit minimal.

Une fois ces étapes réalisées, le réseau est alors opérationnel, et il est possible de l'appliquer alors pour avoir le morceau voulu.

Chapitre 3

Programmation du réseau de neurones

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.

Sources

1. https ://www.youtube.com/watch ?v=KVNhk6uGmr8 : Réseaux de neurones : introduction et applications par Joseph Ghafari
2. http ://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/
3. Apprentissage statistique - Gerard Dreyfus
4. Réseaux de Neurones Artificiels - Manuel Clergue (Université de Nice)
5. https ://medium.com/towards-data-science/can-a-deep-neural-network-compose-music-f89b6ba4978d