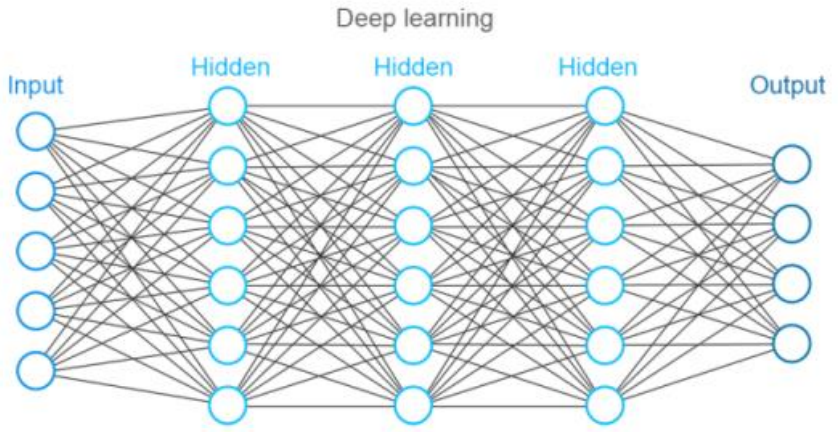
L'apprentissage par réseau de neurones – Partie 1



https://www.altexsoft.com/whitepapers/fraud-detection-how-machine-learning-systems-help-reveal-scams-in-fintech-healthcare-and-ecommerce/

Introduction

Pour réaliser ce laboratoire, vous avez accès au ***projet Java* SIM**. Vous pouvez télécharger le projet avec lien suivant :

http://physique.cmaisonneuve.qc.ca/svezina/projet/apprentissage_reseau/download/SIM-ApprentissageReseau.zip

Décompressez le fichier « *SIM-ApprentissageReseau.zip* **»** dans un répertoire « *java* » qui vous permettra de définir votre ***workspace*** lors de l'ouverture du logiciel ***Eclipse***.

À l'ouverture du logiciel ***Eclipse***, dans l'onglet ***File***, choisissez ***Switch Workspace*** et identifiez la localisation de votre répertoire de projet « *java* ». Dans l'onglet ***File***, choisissez ***New*** et prenez ***New Java Project***. Dans la boîte d'édition ***Project name***, entrez le nom de projet *SIM* (le nom du répertoire du projet contenu dans le fichier *SIM-ApprentissageReseau.zip*). Vous avez maintenant configuré votre environnement de développement.

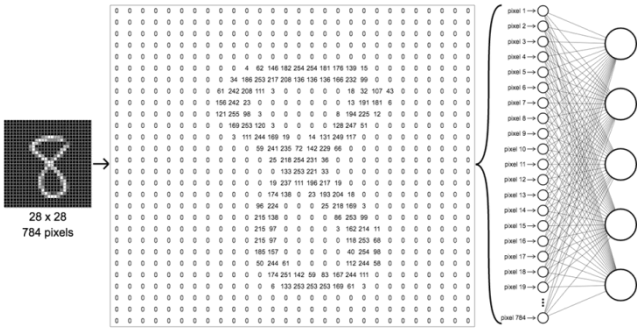
Durant le laboratoire, vous devrez répondre à des questions conceptuelles. Vous pouvez télécharger le cahier de réponse avec lien suivant :

http://physique.cmaisonneuve.qc.ca/svezina/projet/apprentissage_reseau/download/Cahier-ApprentissageReseauPartie1.docx

La description générale du programme

Un **réseau de neurones** est une structure de données qui reproduit le comportement d'une **fonction mathématique**. Les valeurs de sortie du réseau dépendront de ce que l'on désire reproduire.

Par exemple, un réseau calibré[1] peut prédire ce que représente une image. Dans l'illustration ci-contre, le réseau a été conçu pour reconnaître si une image représente le chiffre « 0 », « 1 », … , « 8 », « 9 ». Ce réseau comporte alors 10 sorties admissibles. Bien que cette tâche semble simple pour un humain, écrire un programme réalisant cette tâche est très difficile, car la fonction mathématique réalisant cette tâche n'est pas connue.



http://lucenaresearch.com/deep-neural-networks/

Illustration d'une image convertie en vecteur d'entrée (*input*) pour un réseau de neurones.

Grâce à un entraîneur de réseau, on peut utiliser 60 000 images dont la réponse a été déterminée par un humain et enseigner à notre réseau à les reconnaître. **La descente du gradient** représente la technique itérative qui permettra de calibrer les paramètres du réseau afin que celui-ci reproduise la fonction mathématique désirée.

Le but de ce laboratoire sera d'implémenter des instructions mathématiques requises pour réaliser cette tâche complexe dans le but de pouvoir calibrer un réseau par l'apprentissage et ainsi développer une forme **d'intelligence artificielle**.

La première partie de ce laboratoire consistera à réaliser l'activation d'un réseau de neurones et d'étudier son comportement en comparant un réseau non calibré (sans apprentissage) avec un réseau calibré (avec apprentissage). La deuxième partie de ce laboratoire consistera à réaliser la descente du gradient et de calibrer des réseaux par l'apprentissage.

[1] Nous pouvons utiliser ce synonyme lorsque l'apprentissage a été réalisé par le réseau.

L'architecture

Afin de vous faire mieux comprendre les liens entre les différents composants qui constituent un réseau de neurones, voici l'architecture retenue pour implémenter les fonctionnalités du réseau :

- Réseau de neurones de Classe utilitaire réalisant des calculs de plusieurs fonctions mathématique lié aux réseaux de neurones (**SNeuralMath**)

  Paramètres : Aucun

  Méhodes : Plusieurs fonctions mathématique.

- Réseau de neurones (**SNetwork**)

  Paramètres : Couches de neurones organisées en tableau (**SLayer**[ ])

  Méthodes : Activation des couche du réseau (return double[ ])
  Propagation de l'erreur dans les couches du réseau (return double[ ])

- Couche de neurones de type connection complète (**SFullConnectedLayer** ***extends*** **SLayer**)

  Paramètres : Fonction d'agrégation $z_u^{(k)}$ (**SAggregationFunction**)
  Fonction d'activation $a_u^{(k)}$ (**SActivationFunction**)
  Les poids $w_{uv}^{(k)}$ (**SWeights**, contenant une matrice)
  Les biais $b_u^{(k)}$ (**SBias**, contenant un vecteur)

  Méthodes : Activation de la couche par l'activation en séquence de ses fonctions (return double[ ])
  Propagation de l'erreur dans la couche (return double[ ])

- Fonction neuronale abstraite qui fait la gestion de la mémoire des données (**SMemoryNeuralFunction**)
  Paramètres : forward_inputs, accès sur l'entrée des données $a_v^{(k-1)}$ ou $z_u^{(k)}$ (double[ ])
  forward_outputs, propagation des donnée $z_u^{(k)}$ ou $a_u^{(k)}$ (double[ ])
  backward_inputs, accès sur l'entrée de l'erreur $\Delta_u^{(k+1)}$ ou $\Delta_u^{a(k)}$ (double[ ])
  backward_outputs, propagation de l'erreur $\Delta_v^{z(k)}$ ou $\Delta_u^{a(k)}$ (double[ ])

- Fonction neuronale abstraite qui fait la gestion de l'activation et la propagation des erreur (**SAbstractNeuralFunction** ***extends*** **SMemoryNeuralFunction**)

  Paramètres : aucun
  Méthodes : Activation de la fonction de façon générale (return double[ ])
  Propagation de l'erreur de façon générale (return double[ ])

- Fonction d'agrégation (**SAggregationFunction** ***extends*** **SAbstractNeuralFunction**)

  Paramètres : Accès sur les poids $w_{uv}^{(k)}$ (**SWeights**)
  Accès sur les biais $b_u^{(k)}$ (**SBias**)

  Méthodes : Agrégation de la fonction $z_u^{(k)}$ (return double[ ])
  Propagation de l'erreur $\Delta_v^{z(k)}$ (return double[ ])

- Fonction d'activation (**SActivationFunction** ***extends*** **SAbstractNeuralFunction**)

  Paramètres : Aucun

  Méthodes : Activation de la fonction $a_u^{(k)}$ (return double[ ])
  Propagation de l'erreur $\Delta_u^{a(k)}$ (return double[ ])

Cette architecture permettra l'activation d'un réseau dans la cascade suivante d'activation :

En séquence de couche
$a^{(*)} \rightarrow Couche^{(0)} \rightarrow Couche^{(1)} \rightarrow ... \rightarrow Couche^{(k)} \rightarrow Couche^{(k+1)} \rightarrow ... \rightarrow Couche^{(L)} \rightarrow a^{(L)}$
En séquence de fonction agrégation-activation
$a^{(*)} \rightarrow z^{(0)} \rightarrow a^{(0)} \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow ... \rightarrow z^{(k)} \rightarrow a^{(k)} \rightarrow z^{(k+1)} \rightarrow a^{(k+1)} \rightarrow ... \rightarrow z^{(L)} \rightarrow a^{(L)}$

où $a^{(*)}$ est un vecteur d'entrée au réseau (la donnée à activer par le réseau) et $a^{(L)}$ est le vecteur de sortie du réseau correspondant à l'activation du réseau.

Lors de la propagation du gradient, cette architechture permettra la propagation de l'erreur dans le réseau sous la cascade suivante :

En séquence de couche
$C \rightarrow couche^{(L)} \rightarrow ... \rightarrow couche^{(k+1)} \rightarrow couche^{(k)} \rightarrow ... \rightarrow couche^{(1)} \rightarrow couche^{(0)} \rightarrow \Delta^{(*)}$
En séquence de fonction agrégation-activation à partir de la fonction d'erreur
$C \rightarrow \Delta^{C(L)} \rightarrow \Delta^{a(L)} \rightarrow \Delta^{z(L)} \rightarrow ... \rightarrow \Delta^{a(k+1)} \rightarrow \Delta^{z(k+1)} \rightarrow \Delta^{a(k)} \rightarrow \Delta^{z(k)} \rightarrow ... \rightarrow \Delta^{a(1)} \rightarrow \Delta^{z(1)} \rightarrow \Delta^{a(0)} \rightarrow \Delta^{z(0)}$

où C est une fonction d'erreur qui va comparer l'activation du réseau $a^{(L)}$ avec un vecteur de référence y afin de valider par comparaison si l'activation du réseau est bonne ou mauvaise ce qui permettra d'entraîner le réseau. Le terme $\Delta^{(*)}$ correspond à l'erreur propagée à l'entrée du réseau s'il y a plusieurs réseaux en séquence activation (ce qui ne sera pas le cas pour nous).

1.1 – La première exécution du programme

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : aucun

Prérequis : aucun

Dans la fenêtre ***Package Explorer*** du logiciel ***Eclipse***, ouvrez le répertoire **SIM** puis ouvrez le répertoire **src**. Constatez la présence de quelques ***packages*** nécessaires à l'exécution de ce programme. Ouvrez le ***package*** **sim.application** et lancez l'exécution de la classe **SIMLearning.java** à l'aide d'un « clic droit » sur le fichier. Sélectionnez dans le ***pop-pop menu*** l'option ***Run As*** et l'autre option ***Java Application***.

Choisissez une option parmi les divers choix admissibles. Présentement, le programme ne réaliser pas grand-chose, car l'application ne possède pas encore toutes ses fonctionnalités. Tout au long du laboratoire, vous devrez implémenter des méthodes vous permettant de traiter différentes situations comme les suivantes :

- Effectuer un calcul d'activation et de propagation d'erreur (celui réalisé en prélaboratoire)
- Breast Cancer Wisconsin (Diagnostic) Data Set
- MNIST Handwritten Digits Data Set

1.2 – L'exécution des JUnit Test

Fichier à modifier : aucun

Prérequis : aucun

Pour s'assurer de la qualité d'un programme, il est important de tester les fonctionnalités des différentes méthodes implémentées. L'environnement de développement ***Eclipse*** permet l'exécution de batterie de tests unitaires avec une gestion des succès (*succes*) et des échecs (*fail*).

Un test unitaire réalise l'exécution d'une méthode (ou quelques méthodes) d'une classe ou de plusieurs classes dans un scénario particulier. Le résultat de l'exécution (« ***calculated*** ») est alors comparé avec un résultat attendu (« ***expected*** »). Si le résultat calculé est identique au résultat attendu, le test est un succès. Dans le cas contraire, le test est alors un échec.

https://www.javacodegeeks.com/2013/12/parameterized-junit-tests-with-junitparams.html

Afin de vous familiariser avec l'exécution des ***JUnit Test***, vous allez réaliser l'exécution de l'ensemble des tests préalablement écrit pour vous afin de vous guider dans la qualité de vos implémentations.

Pour ce faire, vous allez :

- Dans la fenêtre ***Package Explorer***, ouvrez le répertoire **SIM** contenant l'ensemble des éléments du projet.
- Sur le répertoire **test/src**, effectuez un « clic droit » et sélectionnez dans le pop-pop menu l'option ***Run As*** et réalisez l'exécution de type ***JUnit Test***.
- (**Si les étapes précédentes ne fonctionnent pas**) Établir le lien avec la librairie ***JUnit 4***. Faire un clic droit sur le répertoire **SIM** dans la fenêtre ***Package Explorer*** et choisir l'option ***Properties***. Dans la propriété ***Java Build Path***, choisir l'onglet ***Libraries*** et activer le bouton ***Add Library***. Choisir ***JUnit*** et activer le bouton ***next***. Choisir ***JUnit 4*** et activer le bouton ***Finish***.

Dans la fenêtre ***JUnit***, vous pouvez visualiser l'ensemble des tests effectués pour valider certaines fonctionnalités du projet :

- Si toutes les implémentations sont adéquates, une **couleur verte** sera affichée (Failures : 0).
- S'il y a des implémentations inadéquates, une **couleur rouge** sera affichée (Failures : « nombre > 0 »).

Dans la fenêtre ***Console***, vous remarquerez la présente de messages indiquant que certains tests n'ont pas été effectués, mais qu'ils sont considérés comme des succès. Ce sont des tests reliés à des méthodes que vous devrez implémenter. Ces méthodes retournent présentement une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Lorsque l'implémentation sera réalisée, le test sera effectué « officiellement ».

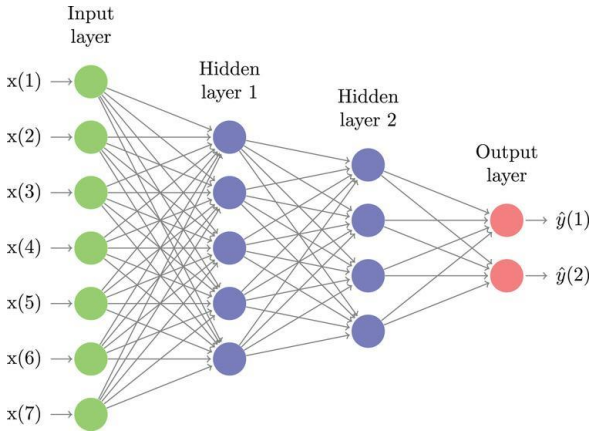
Pour lancer l'exécution d'un nombre restreint de tests, vous n'avez qu'à effectuer le « clic droit » sur le répertoire ou le fichier désiré. Par exemple, vous pouvez lancer le test de la classe **SMathTest** du package ***sim.math*** situé dans le répertoire ***test/src/math***. Vous remarquerez qu'il est en succès (**couleur verte**), mais qu'il y a des tests non effectué.

La fonction d'agrégation *z*

La fonction d'agrégation consiste à regrouper l'information des neurones d'une couche $k-1$ afin de les intégrer comme nouvelle information à chacun des neurones de la couche k.

Pour ce faire, la couche k utilisera ses propres informations comme les poids $w_{uv}^{(k)}$ (*weight*) et ses biais $b_u^{(k)}$ (*bias*) qu'elle combinera avec l'activation $a_v^{(k-1)}$ de la couche précédente $k-1$ afin de réaliser un calcul d'agrégation $z_u^{(k)}$ pour chacun de ses neurones. Le calcul réalisé dépendra du type de fonction d'agrégation employée par la couche k.

L'un des fonctions les plus populaires est la somme pondérée (*weighted sum*) et c'est cette fonction que vous allez implémenter dans ce programme.



https://content.iospress.com/articles/algorithmic-finance/af176

Illustration d'un réseau de neurones à 2 couches cachées. Les connections entre les neurones vert et chacun des neurones bleu représente une agrégation des données de la couche précédente avec les poids de la couche à activer.

Mathématiquement, la somme pondérée peut être représentée comme le produit d'une matrice (les poids $w_{uv}^{(k)}$) avec un vecteur (les entrées $a_v^{(k-1)}$) avec l'ajout d'un autre vecteur (les biais $b_u^{(k)}$) :

$$\begin{pmatrix} z_0^{(k)} \\ z_1^{(k)} \\ \dots \\ z_u^{(k)} \\ \dots \\ z_G^{(k)} \end{pmatrix} = \begin{pmatrix} w_{00}^{(k)} & w_{01}^{(k)} & \dots & w_{0v}^{(k)} & \dots & \dots & w_{0F}^{(k)} \\ w_{10}^{(k)} & w_{11}^{(k)} & \dots & w_{1v}^{(k)} & \dots & \dots & w_{1F}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{u0}^{(k)} & w_{u1}^{(k)} & \dots & w_{uv}^{(k)} & \dots & \dots & w_{uF}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{G0}^{(k)} & w_{G1}^{(k)} & \dots & w_{Gv}^{(k)} & \dots & \dots & w_{GF}^{(k)} \end{pmatrix} \begin{pmatrix} a_0^{(k-1)} \\ a_1^{(k-1)} \\ \dots \\ a_v^{(k-1)} \\ \dots \\ \dots \\ a_F^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_0^{(k)} \\ b_1^{(k)} \\ \dots \\ b_u^{(k)} \\ \dots \\ b_G^{(k)} \end{pmatrix}$$

tel que $$z_u^{(k)} = \sum_{v=0}^{N^{(k-1)}-1} w_{uv}^{(k)} a_v^{(k-1)} + b_u^{(k)}$$

où $F = N^{(k-1)} - 1$: Indice du dernier neurone en entrée (indice de colonne de la matrice).

$G = N^{(k)} - 1$: Indice du dernier neurone en sortie (indice de ligne de la matrice).

Dans l'architechture de ce programme, toutes les fonctions d'agrégations (comme la somme pondérée) comportent l'héritage suivant :

SAbstractNeuralFunction (allocation de l'espace mémoire des neurones : **double**[ ] forward_outputs)

↦ **SAggregationFunction** (définition des fonctionnalités et accès aux poids W et biais B)

↦ **SWeightedSumFunction** (implémentation de la somme pondérée)

↦ **SQuadricSumFunction** (fonction non traitée dans ce labo)

Cette hiérarchie de classe permettra à toutes les fonctions d'agrégation d'avoir accès aux champs forward_inputs (entrée de la fonction) et forward_outputs (sortie de la fonction) de la fonction nécessaire à la phase « d'activation » du réseau. Cette structure permettra l'automatisation des calculs pour tous les neurones d'une fonction lorsque la méthode forward( ) aura été implémentée dans chacune des classes (méthode réalisant le calcul pour un seul neurone).

2.1 Le produit scalaire

Fichier à modifier : **SArrays.java**

Prérequis : aucun

Dans la classe **SArrays** disponible dans le ***package* sim.util**, vous allez programmer la méthode suivante :

public static double dot(**double**[ ] A, **double**[ ] B) **throws** ArrayIndexOutOfBoundsException

L'objectif de cette méthode sera de réaliser le calcul du produit scalaire :

$$C = \vec{A} \cdot \vec{B} = \sum_{i=0}^{N-1} A_i B_i$$

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction **throw new** SNoImplementationException( ); et complétez l'implémentation en utilisant chaque élément des tableaux A et B comme étant les composantes A_i et B_i des vecteurs $\vec{A}$ et $\vec{B}$ et retournez le résultat du produit scalaire.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SArraysTest** du ***package* sim.util** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 2.1 :

Soit deux vecteurs $\vec{A}$ et $\vec{B}$ non nul, sous quelle condition le résultat du produit scalaire entre ces deux vecteurs sera égal à zéro ?

2.2 La somme pondérée

Fichier à modifier : **SNeuralMath.java**

Prérequis : 2.1

Dans la classe **SNeuralMath** disponible dans le ***package* sim.nn.function**, vous allez programmer la méthode suivante :

public static double weightedSum(**int** u, **double**[ ] a, SBufferedMatrix w, **double**[ ] b)

L'objectif de cette méthode sera d'appliquer la fonction d'agrégation de la somme pondérée pour un neurone. Cette méthode devra retourner l'agrégation $z_u^{(k)}$ d'un neurone u appartenant à la couche k d'un réseau en utilisant le vecteur d'entrées $a_v^{(k-1)}$ (a) de la couche $k-1$ avec les les poids $w_{uv}^{(k)}$ (w) et le biais $b_u^{(k)}$ (b) de la couche k du réseau.

Mathématiquement le calcul à réaliser correspond au produit scalaire entre le vecteur $a_v^{(k-1)}$ et la ligne u de la matrice $w_{uv}^{(k)}$ auquel on ajoute la valeur du biais $b_u^{(k)}$ d'indice u :

$$z_u^{(k)} = \sum_{v=0}^{N^{(k-1)}-1} w_{uv}^{(k)} a_v^{(k-1)} + b_u^{(k)}$$

(Notation des poids en matrice ligne-colonne)

$$\begin{pmatrix} z_0^{(k)} \\ z_1^{(k)} \\ \dots \\ z_u^{(k)} \\ \dots \\ z_G^{(k)} \end{pmatrix} = \begin{pmatrix} w_{00}^{(k)} & w_{01}^{(k)} & \dots & w_{0v}^{(k)} & \dots & \dots & w_{0F}^{(k)} \\ w_{10}^{(k)} & w_{11}^{(k)} & \dots & w_{1v}^{(k)} & \dots & \dots & w_{1F}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{u0}^{(k)} & w_{u1}^{(k)} & \dots & w_{uv}^{(k)} & \dots & \dots & w_{uF}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{G0}^{(k)} & w_{G1}^{(k)} & \dots & w_{Gv}^{(k)} & \dots & \dots & w_{GF}^{(k)} \end{pmatrix} \begin{pmatrix} a_0^{(k-1)} \\ a_1^{(k-1)} \\ \dots \\ a_v^{(k-1)} \\ \dots \\ \dots \\ a_F^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_0^{(k)} \\ b_1^{(k)} \\ \dots \\ b_u^{(k)} \\ \dots \\ b_G^{(k)} \end{pmatrix}$$

(Représentation où $u \in [0, G]$ et $v \in [0, F]$)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation en utilisant la méthode

public static double dot(**double**[ ] A, **double**[ ] B)

de la classe **SArrays** pour réaliser le produit scalaire entre les poids $w_{uv}^{(k)}$ de la ligne u de la matrice des poids $w^{(k)}$ et le vecteur d'entrée $a_v^{(k-1)}$.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMathTest** du ***package*** **sim.nn.function** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 2.2 :

Dans le processus de la classification des données d'un réseau de neurones, à quelle forme géométrique compare-t-on la fonction d'agrégation de la somme pondérée ?

(FACULTATIF) Si vous le désirez, consultez la classe **SWeightedSumFunction** disponible dans le ***package*** **sim.nn.function.aggregation** afin de constater que la méthode weightedSum(…) a été introduite pour vous dans la méthode :

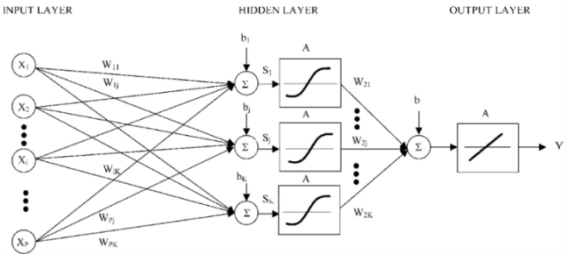
protected double forward(**int** u)

Ceci permettra à la fonction d'agrégation (de classe **SWeightedSumFunction**) d'utiliser cette implémentation pour évaluer l'agrégation du neurone u. Le mot « forward » fait référence à l'action de faire progresser vers l'avant les donnée dans la fonction ce qui est nécessaire lors de l'activation du réseau.

La fonction d'activation *A*

La fonction d'activation consiste à transformer l'agrégation $z_u^{(k)}$ d'un neurone u d'une couche k afin de recadrer sa valeur à l'intérieur d'un intervalle bien défini qui portera le nom d'activation $a_u^{(k)}$.

Par exemple, une fonction d'activation peut avoir le rôle de transformer n'importe quel chiffre en 0 ou 1. Ce comportement permet de convertir un calcul en fonction booléenne (vrai ≡ 1, faux ≡ 0). De plus, la fonction d'activation permet à un réseau de mieux « imiter » le comportement de fonction non linéaire.



https://www.researchgate.net/figure/Feed-forward-neural-network-with-sigmoid-activation-function-X-i-i-1P-input_fig2_273204474

Illustration d'un réseau avec une couche cachée (*HIDDEN LAYER*) ayant la fonction d'activation sigmoïde et une couche de sortie (*OUTPUT LAYER*) ayant la fonction d'activation linéaire.

Puisque la fonction d'agrégation de la somme pondérée est linéaire (de type $z = Ax + b$), la fonction d'activation augmente la « créativité » du réseau dans sa capacité à reproduire un « comportement désiré » comme le comportement d'une fonction non-linéaire

Dans le cadre de ce laboratoire, vous allez implémenter la fonction sigmoïde et tangente hyperbolique. Bien que ces deux fonctions ne sont pas optimales[2], elles sont pédagogiquement intéressantes étudier.

Mathématiquement, l'activation de la somme pondérée peut être représentée de la façon suivante :

$$a^{(k)} = f\left(z^{(k)}\right) = f\left(\begin{pmatrix} w_{00}^{(k)} & w_{01}^{(k)} & \dots & w_{0v}^{(k)} & \dots & \dots & w_{0F}^{(k)} \\ w_{10}^{(k)} & w_{11}^{(k)} & \dots & w_{1v}^{(k)} & \dots & \dots & w_{1F}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{u0}^{(k)} & w_{u1}^{(k)} & \dots & w_{uv}^{(k)} & \dots & \dots & w_{uF}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{G0}^{(k)} & w_{G1}^{(k)} & \dots & w_{Gv}^{(k)} & \dots & \dots & w_{GF}^{(k)} \end{pmatrix} \begin{pmatrix} a_0^{(k-1)} \\ a_1^{(k-1)} \\ \dots \\ a_v^{(k-1)} \\ \dots \\ \dots \\ a_F^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_0^{(k)} \\ b_1^{(k)} \\ \dots \\ b_u^{(k)} \\ \dots \\ b_G^{(k)} \end{pmatrix} \right)$$

$$\text{avec} \quad z_u^{(k)} = \sum_{v=0}^{N^{(k-1)}-1} w_{uv}^{(k)} \, a_v^{(k-1)} + b_u^{(k)}$$

Dans l'architechture de ce programme, toutes les fonctions d'activation (linéaire, sigmoïde, tanh) comportent l'héritage suivant :

SAbstractNeuralFunction (allocation de l'espace mémoire des neurones : **double**[ ] forward_outputs)

↦ **SActivationFunction** (définition des fonctionnalités)

↦ **SLinearFunction** (implémentation de la fonction linéaire, **déjà fait**)

↦ **SSigmoidFunction** (implémentation de la fonction sigmoïde)

↦ **STanhFunction** (implémentation de la fonction tangent hyperbolique)

[2] Les réseaux plus sophistiqués exploitent des fonctions d'activation plus performante pour réaliser des apprentissages plus rapidement.

Cette hiérarchie de classe permettra à toutes les fonctions d'activation d'avoir accès aux champs forward_inputs (entrée de la fonction) et forward_outputs (sortie de la fonction) de la fonction nécessaire à la phase « d'activation » du réseau. Cette structure permettra l'automatisation des calculs lorsque la méthode forward( ) aura été implémentée dans chacune des classes.

3.1 La fonction sigmoïde

Fichier à modifier : **SNeuralMath.java**

Prérequis : auncun

Dans la classe **SNeuralMath** disponible dans le ***package* sim.nn.function**, vous allez programmer la méthode suivante :

```
public static double sigmoid(double x)
```

L'objectif de cette méthode sera de réaliser le calcul de la fonction sigmoïde pour un neurone dont la valeur est égale à *x* :

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

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation et retournez le résultat de la fonction sigmoïde.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMathTest** du ***package* sim.nn.function** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 3.1 :

Historiquement, la fonction sigmoïde a été utilisée dans les réseaux de neurones afin de s'inspirer du comportement biologique d'un réseau neuronal. Quelle était la comparaison biologique associée à la sortie « 0 » de la fonction sigmoïde.

(FACULATIF) Si vous le désirez, consultez la classe **SSigmoidFunction** disponible dans le ***package* sim.nn.function.activation**. Vous y trouverez la fonction sigmoid(**double** x) que vous venez de programmer dans la méthode :

```
protected double forward(int u)
```

Ceci permettra à la fonction neuronale sigmoïde (de classe **SSigmoidFunction**)de calculer l'activation du neurone d'indice u.

3.2 La fonction tangente hyperbolique

Fichier à modifier : **SNeuralMath.java**

Prérequis : auncun

Dans la classe **SNeuralMath** disponible dans le ***package* sim.nn.function**, vous allez programmer la méthode suivante :

```
public static double tanh(double x)
```

L'objectif de cette méthode sera de réaliser le calcul de la fonction tangente hyperbolique pour un neurone dont la valeur est égale à *x* :

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

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation et retournez le résultat de la fonction tangente hyperbolique.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMathTest** du ***package* sim.nn.function** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 3.2 :

Quelle sera la valeur de sortir de la fonction tangente hyperbolique pour une entrée égale à « 0 » ?

(FACULATIF) Si vous le désirez, consultez la classe **STanhFunction** disponible dans le ***package* sim.nn.function.activation**. Vous y trouverez la fonction tanh(**double** x) que vous venez de programmer dans la méthode :

```
protected double forward(int u)
```

Ceci permettra à la fonction neuronale tangente hyperbolique (de classe **STanhFunction**)de calculer l'activation du neurone d'indice u.

3.3 L'activation abstraite d'une fonction sur plusieurs neurones

Fichier à modifier : **SAbstractNeuralFunction.java**

Prérequis : aucun

Dans la classe **SAbstractNeuralFunction** disponible dans le ***package* sim.nn.function**, vous allez programmer la méthode suivante :

```
protected double[ ] forwardPass( )
```

Cette méthode a pour but de généraliser, pour n'importe quel type de fonction neuronale, l'exécution d'une fonction neuronale en mode propagation des données (propagation vers l'avant) pour l'ensemble des neurones.

Vous remarquerez que la méthode forward(**int** u) permettra de réaliser un calcul pour 1 seul neurone de la fonction tandis que la méthode forwardPass( ) permettra de réaliser des calcul pour tous les neurones de la fonction.

Par exemple, pour la fonction d'agrégation de la classe **SWeightedSumFunction** héritant de **SAggregationFunction** qui elle hérite de **SAbstractNeuralFunction**, la méthode forwardPass( ) réalisera le calcul suivant pour tous les neurones :

$$z_u^{(k)} = \sum_{v=0}^{N^{(k-1)}-1} w_{uv}^{(k)} \, a_v^{(k-1)} + b_u^{(k)}$$

où

$$u \in \left[0, N^{(k)} - 1\right] \text{ et } v \in \left[0, N^{(k-1)} - 1\right]$$

avec $G = N^{(k)} - 1$ *et* $F = N^{(k-1)} - 1$

$$\begin{pmatrix} z_0^{(k)} \\ z_1^{(k)} \\ \dots \\ z_u^{(k)} \\ \dots \\ z_G^{(k)} \end{pmatrix} = \begin{pmatrix} w_{00}^{(k)} & w_{01}^{(k)} & \dots & w_{0v}^{(k)} & \dots & \dots & w_{0F}^{(k)} \\ w_{10}^{(k)} & w_{11}^{(k)} & \dots & w_{1v}^{(k)} & \dots & \dots & w_{1F}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{u0}^{(k)} & w_{u1}^{(k)} & \dots & w_{uv}^{(k)} & \dots & \dots & w_{uF}^{(k)} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ w_{G0}^{(k)} & w_{G1}^{(k)} & \dots & w_{Gv}^{(k)} & \dots & \dots & w_{GF}^{(k)} \end{pmatrix} \begin{pmatrix} a_0^{(k-1)} \\ a_1^{(k-1)} \\ \dots \\ a_v^{(k-1)} \\ \dots \\ \dots \\ a_F^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_0^{(k)} \\ b_1^{(k)} \\ \dots \\ b_u^{(k)} \\ \dots \\ b_G^{(k)} \end{pmatrix}$$

Également, pour la fonction d'activation de la classe **STanhFunction** héritant de **SActivationFunction** qui elle hérite de **SAbstractNeuralFunction**, la méthode forwardPass( ) réalisera le calcul suivant pour tous les neurones :

$$a_u^{(k)} = \tanh\left(z_u^{(k)}\right)$$

où

$$u \in \left[0, N^{(k)} - 1\right]$$

avec $G = N^{(k)} - 1$

$$\begin{pmatrix} a_0^{(k)} \\ a_1^{(k)} \\ \dots \\ a_u^{(k)} \\ \dots \\ a_G^{(k)} \end{pmatrix} = \begin{pmatrix} \tanh\left(z_0^{(k)}\right) \\ \tanh\left(z_1^{(k)}\right) \\ \dots \\ \tanh\left(z_u^{(k)}\right) \\ \dots \\ \tanh\left(z_G^{(k)}\right) \end{pmatrix}$$

Pour ce faire, vous devrez utiliser la méthode abstraite

abstract protected double forward(**int** u)

disponible dans la classe **SAbstractNeuralFunction** réalisant la propagation des données vers l'avant pour le neurone u.

Vous remarquerez que même si la méthode forward(**int** u) n'est pas définie dans cette classe (***abstract class***), vous aurez quand même le droit de l'utiliser dans cette classe. Si une classe hérite de **SAbstractNeuralFunction** (ex : **SWeightedSumFunction**), elle devra nécessairement implémenter la méthode forward(**int** u) (comme vous l'avez consulté dans les 2 étapes précédentes). Ceci définira le calcul à réaliser par la méthode forward(**int** u) et rendra le calcul de la méthode forwardPass( ) concret.

« C'est ça la beauté d'un langage orienté objet supportant l'héritage! »

Pour réaliser votre implémentation, utilisez le champ local de la classe

protected double[ ] forward_outputs

correspondant à la sortie de la fonction en mode vers l'avant pour enregistrer vos calculs réalisés par la méthode forward(**int** u) pour chacun des u neurones.. Lorsque tous vos neurones auront été calculés et enregistrés, retournez votre tableau des résultats avec l'instruction

return this.forward_outputs

Exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SAbstractNeuralFunctionTest** du ***package*** **sim.nn.function** située dans le répertoire **test/src**. Ces tests utiliserons d'autres fonctions afin de réaliser des tests sans méthode abstraite. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 3.3 :

Dans l'équation

$$z_u^{(k)} = \sum_{v=0}^{N^{(k-1)}-1} w_{uv}^{(k)} a_v^{(k-1)} + b_u^{(k)}$$

de la somme pondérée, donnez une interprétation géométrique dans le fait d'introduire le terme $b_u^{(k)}$ dans l'expression de celle-ci.

(FACULATIF) La fonctionnalité forwardPass( ) a été introduite pour vous dans la méthode

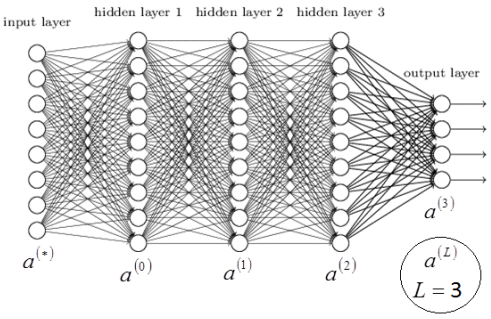
public double[ ] forwardPropagation(**double**[ ] inputs) **throws** IllegalArgumentException

afin de permettre à toutes les fonctions neuronales de pouvoir réaliser la propagation des données à partir de données d'entrée (inputs) dans la fonction.

L'activation d'un réseau

L'activation d'un réseau consiste introduire un vecteur d'entrée $a^{(*)}$ (*input layer*) dans un réseau afin d'y réaliser des calculs à partir de ce vecteur et des paramètres du réseau comme les poids $w^{(k)}$ et biais $b^{(k)}$. Le résultat obtenu devient alors l'activation $a^{(L)}$ du réseau étant l'activation de la dernière couche (*output layer*).

Pour ce faire, on réalise des activations couche par couche. Le vecteur $a^{(*)}$ est utilisé pour calculer l'activation $a^{(0)}$ de la 1re couche (*hidden layer 1*). Le vecteur $a^{(0)}$ est utilise pour calculer l'activation $a^{(1)}$ de la 2e couche (*hidden layer 2*). Après le calcul de l'activation $a^{(k-1)}$, on utilise ce résultat pour réaliser l'activation $a^{(k)}$. Il s'en suit une cascade de calcul jusqu'à obtenir l'activation du réseau $a^{(L)}$.



https://datawarrior.wordpress.com/2017/10/31/interpretability-of-neural-networks/
(Modifié par Simon Vézina)

Illustration d'un réseau de neurones de type « full connected » à 4 couches.

4.1 L'activation d'une couche de neurones

Fichier à modifier : **SFullConnectedLayer.java**

Prérequis : 2.2, 3.1 et 3.3

Dans la classe **SFullConnectedLayer** disponible dans le ***package*** **sim.nn.layer**, vous allez programmer la méthode suivante :

```
public double[ ] forwardPropagation(double[ ] input) throws IllegalArgumentException
```

Cette méthode a pour but d'exécuter l'activation de la couche de neurones (propagation des données vers l'avant) ayant une fonction d'agrégation Z et une fonction d'activation A.

Dans cette classe, on y retrouve les champs suivantes :

`protected final SWeights W;`	(Les poids de la couche de neurones)
`protected final SBias B;`	(Les biais de la couche de neurones)
`protected final SAggregationFunction Z;`	(La fonction d'agrégation de la couche)
`protected final SActivationFunction A;`	(La fonction d'activation de la couche)

Pour réaliser l'activation de la couche, vous devrez :

1) Évaluer l'agrégation $z^{(k)}$ de la couche de neurones en utilisant l'objet Z représentant la fonction d'agrégation. L'appel de la méthode

   ```
   public double[ ] aggregation(double[ ] inputs, SWeights W, SBias B)
   ```

   permet de calculer $z^{(k)}$.
2) Évaluer l'activation $a^{(k)}$ de la couche de neurones en utilisant l'objet A représentant la fonction d'activation. L'appel de la méthode

   ```
   public double[ ] activation(double[ ] Z)
   ```

   permet de calculer $a^{(k)}$.
3) Retourner le résultat de l'activation $a^{(k)}$.

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation en utilisant les méthodes appartenant aux objets **this**.Z et **this**.A pour retourner l'activation de la couche.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SFullConnectedLayerTest** du ***package*** **sim.nn.layer** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

4.2 Retour sur le prélaboratoire, partie 1 (activation d'une couche)

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : aucun

Prérequis : 4.1

Afin de valider votre prélaboratoire à l'aide des fonctionnalité que vous avez intégré dans le système, vous allez exécuter le programme **SIMLearning** disponible dans le ***package*** **sim.application**.

Choisissez l'option **« Prelaboratoire – Partie 1 (Activation de la couche) »** afin de comparer l'exécution de JAVA et les calculs que vous avez réalisés dans votre prélaboratoire.

Constatez que dans l'affichage texte, on y retrouve les informations suivantes :

Poids *W* de la couche	Biais B de la couche	Données *x* de la couche
$W = \begin{pmatrix} 1.0 & 4.0 \\ 3.0 & 2.0 \\ -2.0 & -3.0 \end{pmatrix}$	$b = \begin{pmatrix} 0.8 \\ 0.5 \\ -0.2 \end{pmatrix}$	$x = \begin{pmatrix} -0.4 \\ 0.6 \end{pmatrix}$

Répondez aux questions suivantes dans le cahier de réponse :

Question 4.2.1 :

Quel est le résultat de l'activation de votre réseau ?

Question 4.2.2 :

Est-ce que l'activation de votre réseau sur JAVA correspond à la réponse de votre prélaboratoire ?

(FACULATIF) Si vous le désirez, vous pouvez consulter la méthode

```
public static void prelaboratoirePartie1( )
```

de la classe **SIMLearning** afin de comprendre comment il faut écrire le « code » afin de réaliser le calcul de votre prélaboratoire.

4.3 L'activation d'un réseau de neurones

Fichier à modifier : **SNetwork.java**

Prérequis : 4.1

Dans la classe **SNetwork** disponible dans le ***package*** **sim.nn.network**, vous allez programmer la méthode suivante :

```
public double[ ] forwardPropagation(double[ ] input) throws IllegalArgumentException
```

Cette méthode a pour but d'exécuter l'activation du réseau (propagation des données vers l'avant) en appliquant en séquence l'activation des multiples couches décrivant le réseau.

Dans cette interface, on peut avoir accès à un tableau des couches de neurones du réseau à l'aide de l'instruction

```
public SLayer[ ] getLayers( );
```

(Tableau contenant la séquence des couches du réseau en ordre d'activation)

L'ordre des couches dans le tableau correspond à l'ordre de l'activation des couches dans le réseau.

Pour réaliser l'activation du réseau, vous devrez :

1) Obtenir le tableau des couches (vous pouver le nommer layers) de type SLayer[ ] à l'aide de l'instruction getLayers( ).
2) Réaliser en séquence l'activation $a^{(k)}$ de toutes les couches du réseau à partir du tableau des couches. Débutez l'activation par la couche $k=0$. Itérez jusqu'à la dernière couche du réseau $k=L$ où L correspond à l'indice de la dernière couche du réseau. Vous pouvez obtenir L par l'instruction **this**.getL( ). N'oubliez pas que l'activation de la couche k devient le vecteur d'entrée pour la couche $k+1$.
3) Retournez le résultat de l'activation $a^{(L)}$ de la dernière couche.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNetworkTest** du ***package*** **sim.nn.network** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

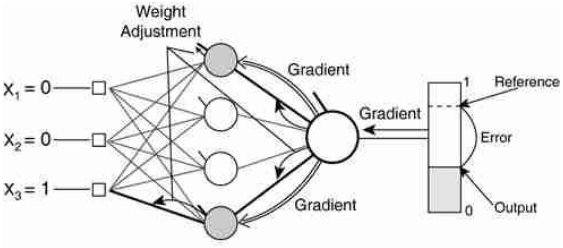
Répondez aux questions suivantes dans le cahier de réponse :

Question 4.3 :

Dans un réseau tel qu'étudié dans ce laboratoire, un réseau de dimension 4 x 3 x 2 aura besoin de combiens de paramètres (poids et biais) ?

La fonction d'erreur *C*

La fonction d'erreur[3] C consiste à évaluer à quel point il y a une différence entre l'activation d'un réseau $a^{(L)}$ calculée à partir d'un vecteur d'entrée $a^{(*)}$ et une valeur attendue y. C'est à partir de la fonction d'erreur qu'il est possible d'évaluer si la prédiction d'un réseau est bonne ou non (la classification est respectée). C'est également à partir de cette fonction qu'on pourra pondérer la modification à apporter au réseau afin que celui-ci s'améliore par un processus d'apprentissage. Le réseau pourra ainsi réduire son erreur lors de la prochaine activation du vecteur d'entrée $a^{(*)}$.



https://mc.ai/hi-iam-vignesh-and-this-is-my-first-blog-on-neural-network-on-topic-forward-propagation-and-back/

Illustration de l'activation d'un réseau à partir d'un réseau à une couche comportant 3 neurones d'entrée (x_1, x_2 et x_3). La fonction d'erreur détermine l'écart entre la sortie (*Output*) et la valeur attendue (*Reference*) et l'on utilise cette information pour calcul l'ajustement des poids/biais du réseau (*Weight Adjustement*) à l'aide d'un gradient (*Gradient*).

L'une des fonctions d'erreur populaire est la fonction d'erreur quadratique qui porte également le nom de fonction d'énergie (*energy*). C'est cette fonction que vous allez implémenter dans ce laboratoire.

Dans l'architechture de ce programme, toutes les fonctions d'erreur respecte la hiérarchie suivante :

SAbstractNeuralFunction (allocation de l'espace mémoire des neurones : **double**[ ] forward_outputs)

↦ **SErrorFunction** (définition des fonctionnalités)

↦ **SLinearErrorFunction** (implémentation de la fonction d'erreur linéaire, déjà fait)

↦ **SEnergyErrorFunction** (implémentation de la fonction d'erreur énergie)

[3] Cette fonction porte également plusieurs autres nom : coût (*cost*), perte (*loss*)

5.1 L'erreur de la fonction Énergie

Fichier à modifier : **SNeuralMath**

Prérequis : aucun

Dans la classe **SNeuralMath** disponible dans le ***package* sim.nn.function**, vous allez programmer la méthode suivante :

public static double quadraticError(**double** calculated, **double** expected)

L'objectif de cette méthode sera d'évaluer l'erreur quadratique entre une valeur calculée (*calculated*) et une valeur attendue (*expected*) :

$$C_u = \frac{1}{2}\left(a_u^{(L)} - y_u\right)^2$$

Cette méthode retournera l'erreur C_u à partir de l'activation $a_u^{(L)}$ du neurone u (calculated) et de sa valeur attendue y_u (expected).

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation et retournez l'erreur C_u.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMathTest** du ***package* sim.nn.function** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

(FACULTATIF) Si vous le désirez, consultez la classe **SEnergyErrorFunction** disponible dans le ***package* sim.nn.function.error**. Vous y trouverez la fonction quadraticError (…) que vous venez de programmer dans la méthode :

protected double forward(**int** u)

Ceci permettra à la fonction neuronale énergie (de classe **SEnergyErrorFunction**)de calculer l'erreur du neurone d'indice u.

Le taux de classification

Un réseau de neurones non entraîné est composé de couches de neurones dont la valeur des poids et biais ont été initialisés avec des valeurs réelles aléatoire[4] entre -1 et 1. En réalisant l'activation de ce type de réseau avec un vecteur d'entrée quelconque, le résultat est purement aléatoire. Pour obtenir des bons résultats, il est important d'avoir préalablement entraîné le réseau avec des vecteurs d'entraînement. Cependant, cette tâche sera réalisée ultérieurement dans ce laboratoire.

https://www.linkedin.com/pulse/artificial-intelligence-how-do-neural-networks-work-better-vignali

Un réseau de neurones nécessite en entraînement avec de pouvoir bien reproduire un résultat attendu.

Dans cette partie du laboratoire, vous allez analyser le comportement d'un réseau non entraîné et un réseau préalablement entraîné. Vous allez comparer des **taux de classification** correspondant au pourcentage de « bonne réponse » déterminé par un réseau à partir d'un ensemble de données de validation. Un réseau mal entraîné aura un faible taux et un réseau bien entraîné aura un taux élevé, mais pas nécessairement de 100%.

[4] Initialiser des poids et biais avec des valeurs entre -1 et 1 accélère l'apprentissage.

6.1 L'erreur et la classification d'une donnée

Fichier à modifier : **STrainingAlgorithm.java**

Prérequis : 4.3 et 5.1

Dans la classe **STrainingAlgorithm** disponible dans le ***package*** **sim.nn.trainer**, vous allez programmer la méthode suivante :

```
public static double[ ] computeErrors(SNetwork network, SErrorFunction error_function, SNNData data)
```

Cette méthode a pour but de déterminer l'erreur générée par le réseau lors de l'usage d'une donnée. L'objet data contient plusieurs information : la donnée, le vecteur de la donnée (*vector*), l'activation attendue (*expected*) ainsi que l'enregistrement d'une activation et son erreur.

Pour calculer l'erreur générée par le réseau avec l'usage d'une donnée, vous devrez :

1) Effectuer l'activation du réseau network avec le vecteur (*vector*) correspondant à la donnée data dont l'accès est possible par l'appel data.getVector( ). N'oubliez pas de sauvegarder le résultat de l'activation dans un tableau de type ***double*** (double[ ]).
2) Évaluer l'erreur du réseau avec la fonction d'erreur error_function en utilisant l'activation du réseau et l'activation attendue (*expected*) de la donnée data dont l'accès est possible par l'appel data.getExpected( ). N'oubliez pas de sauvegarder le résultat de l'erreur dans un tableau de type ***double*** (double[ ]).
3) Effectuer la sauvegarde de l'activation du réseau et de l'erreur du réseau relative à une donnée par l'appel la méthode

   ```
   public void setActivationAndErrors(double[ ] activation, double[ ] errors)
   ```

   sur l'objet data (data.setActivationAndErrors(…, …);). Cette sauvegarde permettra à l'entraîneur de réaliser des calculs statistiques sur la performance du réseau sur l'ensemble de ses données.
4) Retourner la valeur de l'erreur du réseau que vous avez calculée (l'étape 2).

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **STrainingAlgorithmTest** du ***package*** **sim.nn.trainer** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

6.2 Breast Cancer Wisconsin (Diagnostic) Data Set

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : aucun

Prérequis : 6.1

Dans la classe **SIMLearning** disponible dans le ***package*** **sim.application**, vous allez analyser la base de données Breast Cancer Wisconsin Data Set. Le programme va analyser des vecteurs à 9 dimensions contenant des valeurs entières de 0 à 10. Les composantes des vecteurs d'entrée $a_k^{(*)}$ correspondent à des caractéristiques obtenues à partir d'images de tumeurs[5] en lien avec la prévention du cancer du sein. Par exemple les composantes peuvent

[5] Tumeur : groupe de cellules anormales qui forment une masse

référence : http://www.cancer.ca/fr-ca/cancer-information/cancer-101/what-is-cancer/types-of-tumours/?region=on)

décrire l'épaisseur de la masse, l'uniformité de la taille de la cellule, etc. L'activation attendue sera $y_0 = 0$ si la tumeur est « bénigne » et $y_0 = 1$ si la tumeur est « cancéreuse ».

Voici un tableau représentant quelques patients diagnostiquées :

	Entrées du réseau									Sortie du réseau
ID	$a_0^{(*)}$	$a_1^{(*)}$	$a_2^{(*)}$	$a_3^{(*)}$	$a_4^{(*)}$	$a_5^{(*)}$	$a_6^{(*)}$	$a_7^{(*)}$	$a_8^{(*)}$	y
…										
1044572	8	7	5	10	7	9	5	5	4	1
1048672	4	1	1	1	2	1	2	1	1	0
1054593	10	5	5	3	6	7	7	10	1	1
…										

Référence : https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

Le réseau utilisé sera de taille **9 x 1** avec la fonction *Z* de somme pondérée (*weigthed sum*) pour l'agrégation et la fonction *A* sigmoïde pour l'activation de l'unique couche du réseau.

Exécutez le programme **SIMLearning** et choisir l'option *(3) Breast Cancer Wisconsin (Diagnostic) Data Set*. Dans l'affichage console, vous trouverez les informations suivantes :

1) Lecture de la base de données (dossier des patients). Il y aura affichage des dossiers non valides.
2) La description d'un réseau aléatoirement initialisé.
3) Le taux de classification de plusieurs réseaux non entraînés.
4) Affichage de quelques activations calculées par le réseau non entraîné.
5) La description d'un réseau entraîné.
6) Le taux de classification d'un réseau entraîné.
7) Affichage de quelques activations calculées réseau entraîné.

Répondez aux questions suivantes dans le cahier de réponse :

Question 6.2.1 :

En moyenne, quel est le pourcentage de la classification du réseau lorsque celui-ci n'est pas entraîné ?

Question 6.2.2 :

En moyenne, quel est le pourcentage de la classification du réseau lorsque celui-ci est adéquatement entraîné ?

Question 6.2.3 :

Sous quelle condition un réseau de neurones dont la classification n'est pas de 100% pourrait être utilisé par une équipe médicale pour diagnostiquer des tumeurs cancéreuses chez des patients ? Justifiez votre réponse en décrivant le type d'erreur qui pourrait être tolérée.

6.3 MNIST Data Set

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : aucun

Prérequis : 6.1

Dans la classe **SIMLearning** disponible dans le ***package* sim.application**, vous allez analyser la base de données MNIST Data Set. Le programme va analyser des vecteurs à 784 dimensions contenant des valeurs réelles entre 0 et 1. Les composantes des vecteurs d'entrée $a_k^{(*)}$ correspondent à une couleur en nuance de gris (0.0 est noir et 1.0 est blanc) pour chacun des pixels d'une image de taille 28 x 28 illustrant un chiffre écrit à la main. Les activations y_k à 10 neurones seront égales à 0 pour tous les neurones k sauf pour celui où l'indice correspond à la signification de l'image. Par exemple, $y = (0,0,1,0,0,0,0,0,0,0)$ sera l'activation recherchée pour une image qui ressemble au chiffre « 2 ».

https://en.wikipedia.org/wiki/File:MnistExamples.png

Échantillons de la base de données MNIST.

Vous pouvez consulter l'illustration (voir page précédente) pour visualiser quelques vecteurs d'entraînement de la base de données MNIST représentés sous la forme d'image.

Le réseau utilisé sera de taille **784 x 16 x 16 x 10** avec la fonction de somme pondérée (*weigthed sum*) pour l'agrégation et la fonction sigmoïde pour l'activation des 3 couches du réseau. Avant d'exécuter le programme, vous devrez installer la collection d'images.

Pour ce faire, vous allez :

1) Télécharger le fichier *mnist_png.zip* au lien

   http://physique.cmaisonneuve.qc.ca/svezina/projet/apprentissage_reseau/download/mnist_png.zip

2) Déplacer le fichier *mnist_png.zip* dans le répertoire */SIM* de votre ***workspace***.
3) Décompresser le fichier *mnist_png.zip*.

Par la suite, exécutez le programme **SIMLearning** et choisir l'option *(4) MNIST Data Set*. Dans l'affichage console, vous trouverez les informations suivantes :

1) Lecture de la base de données (images de validation uniquement).
2) La description d'un réseau aléatoirement initialisé.
3) Le taux de classification de plusieurs réseaux non entraînés.
4) La description d'un réseau entraîné.
5) Le taux de classification d'un réseau entraîné.
6) Affichage de quelques activations calculées réseau entraîné.

Répondez aux questions suivantes dans le cahier de réponse :

Question 6.3.1 :

En moyenne, quel est le pourcentage de la classification du réseau lorsque celui-ci n'est pas entraîné ?

Question 6.3.2 :

En moyenne, quel est le pourcentage de la classification du réseau lorsque celui-ci est adéquatement entraîné ?

Question 6.3.3 :

Pourquoi est-il préférable d'utiliser 10 neurones pour classifier adéquatement les 10 types d'images au lieu d'utiliser qu'un seul neurone de sortie où la valeur numérique correspondrait au type d'image (exemple : une sortie égale à 6.9345 correspondrait à l'image « 7 ») ?

Conclusion

Félicitations ! Vous avez compléter l'implémentation de l'activation de votre réseau de neurones. Vous êtes maintenant près à compléter la 2e partie de ce laboratoire ayant pour but d'entraîner votre réseau de neurones sous différents protocoles d'entraînement.

Remise du programme

Pour effectuer la remise de votre programme, envoyer le fichier ci-dessous sur la **plateforme numérique OMNIVOX/LÉA** dans l'espace de remise prédéterminé par votre enseignant :

- Le **répertoire SIM** de votre projet dans un **format compressé** « zip » sous le nom

  SIM-ApprentissageReseau.zip

  comme vous l'avez initialement téléchargé.