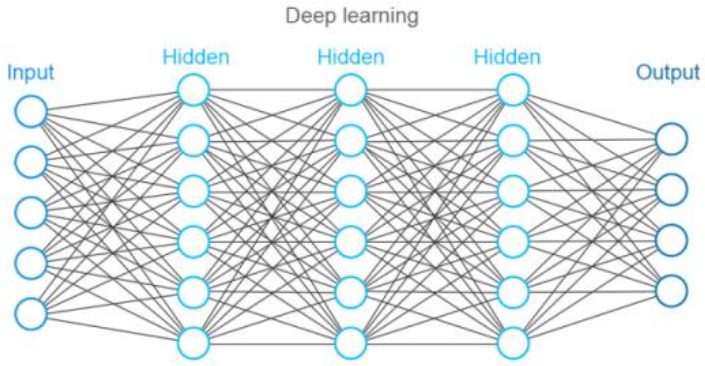
L'apprentissage par réseau de neurones – Partie 2



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

INTRODUCTION **2**

RAPPEL DE L'ARCHITECTURE **2**

LES DÉRIVÉES PARTIELLES DE LA PROPAGATION DU GRADIENT **3**

7.1 La dérivée de la fonction sigmoïde 3
7.2 La dérivée de la fonction tangente hyperbolique 4
7.3 Les dérivées partielles de la fonction d'agrégation *Z* 5
7.4 La dérivée de l'erreur quadratique 6

LA PROPAGATION DE L'ERREUR DE LA PROPAGATION DU GRADIENT **7**

8.1 La propagation de l'erreur de la fonction d'erreur *C* 8
8.2 La propagation de l'erreur de la fonction d'activation *A* 8
8.3 La propagation de l'erreur de la fonction d'agrégation *Z* 9
8.4 La propagation abstraite des erreurs sur plusieurs neurones 9
8.5 La propagation de l'erreur d'une couche de neurones 10
8.6 Retour sur le prélaboratoire, partie 2 (propagation des erreurs) 11
8.7 La propagation de l'erreur d'un réseau 11

LE GRADIENT DE LA FONCTION D'ERREUR **12**

9.1 La dérivée partielle de la fonction d'erreur *C* par rapport au biais *B* 13
9.2 La dérivée partielle de la fonction d'erreur *C* par rapport au poids *W* 13
9.3 Le calcul du gradient de la fonction d'erreur 14

L'APPRENTISSAGE DU RÉSEAU **15**

10.1 L'apprentissage 15

L'ENTRAINEMENT D'UN RÉSEAU **17**

11.1 Le protocole d'entraînement 18
11.2 L'apprentissage de Breast Cancer Wisconsin (Diagnostic) Data Set 18
11.3 L'apprentissage de MNIST Data Set 19

CONCLUSION **20**

REMISE DU PROGRAMME **20**

Introduction

Pour réaliser ce laboratoire, vous devrez réutiliser le code que vous avez développé durant la 1re partie. Ainsi, récupéré votre fichier « *SIM-ApprentissageReseau.zip* **»** et assurez-vous de continuer votre programme dans un répertoire « *java* » qui vous permettra de définir votre ***workspace*** lors de l'ouverture du logiciel ***Eclipse***.

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-ApprentissageReseauPartie2.docx

Rappel de l'architecture

Lors de la 1re partie de ce laboratoire, vous avez travaillé avec les classes suivantes dans le but de réaliser l'activation de votre réseau de neurones :

- **SNeuralMath**
- **SNetwork**
- **SFullConnectedLayer** ***extends*** **SLayer**
- **SAbstractNeuralFunction** ***extends*** **SMemoryNeuralFunction**
- **SAggregationFunction** ***extends*** **SAbstractNeuralFunction**
- **SActivationFunction** ***extends*** **SAbstractNeuralFunction**

Cette architecture vous a permis de réaliser 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
$C \rightarrow \Delta^{C(L)} \rightarrow \Delta^{a(L)} \rightarrow \Delta^{z(L)} \rightarrow ... \rightarrow \Delta^{a(k+1)} \rightarrow \Delta^{z(k+1)} \rightarrow \Delta^{a(k)} \rightarrow \Delta^{z(k)} \rightarrow ... \rightarrow \Delta^{a(1)} \rightarrow \Delta^{z(1)} \rightarrow \Delta^{a(0)} \rightarrow \Delta^{z(0)}$

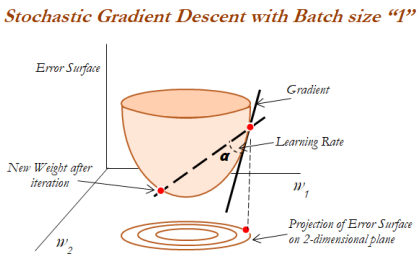
où C est une fonction d'erreur qui va comparer l'activation du réseau $a^{(L)}$ avec un vecteur de référence y afin de valider par comparaison si l'activation du réseau est bonne ou mauvaise ce qui permettra d'entraîner le réseau. Le terme $\Delta^{(*)}$ correspond à l'erreur propagée à l'entrée du réseau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

Les dérivées partielles de la propagation du gradient

La propagation du gradient de la fonction d'erreur $C = C\left(a^{(L)}, y\right)$ permet d'obtenir plusiuers les dérivées partielles

$$\partial C / \partial w_{uv}^{(k)} \quad \text{et} \quad \partial C / \partial b_u^{(k)}$$

par rapport à tous les paramètres du réseau pouvant influencer celle-ci comme les poids $w_{uv}^{(k)}$ et biais $b_u^{(k)}$ nécessaire au calcul de l'activation $a^{(L)}$. Par la suite, cette propagation permettra de définir un gradient ∇C qui sera utilisé pour modifier adéquatement les poids et biais du réseau dans le processus d'apprentissage. En appliquant à répétition ce processus d'apprentissage sur plusieurs vecteurs d'entrée $a^{(*)}$ où une valeur attendue y est associée, on permet au réseau d'apprendre. Cette méthode porte le nom de descente du gradient.



https://www.safaribooksonline.com/library/view/statistics-for-machine/9781788295758/f7c6dca2-593c-449d-8901-b9a98e5fd1c4.xhtml

Illustration de la fonction d'erreur (*Error Surface*) et de la direction qu'il faut emprunter pour modifier les poids (w_1 et w_2) afin de localiser le minimum pour ainsi minimiser l'erreur et améliorer la prédiction du réseau.

Pour obtenir l'ensemble des expressions

$$\partial C / \partial w_{uv}^{(k)} \quad \text{et} \quad \partial C / \partial b_u^{(k)},$$

il y aura beaucoup de petits calculs à réaliser et à assembler. Débutons par évaluer les dérivée partielles suivantes par rapport à nos différentes fonctions d'activations, d'agrégations et d'erreurs :

$$\frac{\partial a_u^{(k)}}{\partial z_u^{(k)}}, \quad \frac{\partial z_u^{(k)}}{\partial w_{uv}^{(k)}}, \quad \frac{\partial z_u^{(k)}}{\partial a_v^{(k-1)}}, \quad \frac{\partial z_u^{(k)}}{\partial b_u^{(k)}} \quad \text{et} \quad \frac{\partial C_u}{\partial a_u^{(L)}}$$

7.1 La dérivée de 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 derivateSigmoid(**double** x)

L'objectif de cette méthode sera de calculer la dérivée de la fonction sigmoïde pour un neurone de valeur *x* :

$$\frac{\partial a_u^{(k)}}{\partial z_u^{(k)}} = \sigma'(x) \qquad \text{tel que} \qquad \sigma'(x) = \frac{\mathrm{d}\,\sigma(x)}{\mathrm{d}x} = \sigma(x) * (1 - \sigma(x)) \text{ avec } \sigma(x) = \frac{1}{1+e^{-x}}$$

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 dérivée de la fonction sigmoïde.

Pour vérifier vos deux implémentations, 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 7.1 :

Lors de l'activation de la fonction sigmoïde avec une grande valeur positive d'entrée, décrivez en mots la valeur que prendra la dérivée de cette fonction correspondant à cette valeur d'entrée.

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

public double dAdZ(**int** u)

Ceci permettra à la fonction neuronale sigmoïde (de classe **SSigmoidFunction**) de calculer la dérivée de son activation $a_u^{(k)}$ pour le neurone d'indice u par rapport à $z_u^{(k)}$ d'indice u.

7.2 La dérivée de 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 derivateTanh(**double** x)

L'objectif de cette méthode sera de calculer la dérivée de la fonction tangent hyperbolique pour un neurone de valeur *x* :

$$\frac{\partial a_u^{(k)}}{\partial z_u^{(k)}} = \tanh'(x) \qquad \text{tel que} \qquad \tanh'(x) = \frac{\mathrm{d}\tanh(x)}{\mathrm{d}x} = 1 - \tanh^2(x) \quad \text{avec} \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

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 dérivée de la fonction tangente hyperbolique.

Pour vérifier vos deux implémentations, 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 7.2 :

Qu'est-ce qu'on en commun la valeur des pentes des fonctions d'activation sigmoïde et tangente hyperbolique pour une valeur d'entrée égale à zéro ?

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

public double dAdZ(**int** u)

Ceci permettra à la fonction neuronale tangente hyperbolique (de classe **STanhFunction**) de calculer la dérivée de son activation $a_u^{(k)}$ pour le neurone d'indice u par rapport à $z_u^{(k)}$ d'indice u.

7.3 Les dérivées partielles de la fonction d'agrégation *Z*

Fichier à modifier : **SWeightedSumFunction.java**

Prérequis : aucun

Dans la classe **SWeightedSumFunction** disponible dans le ***package*** **sim.nn.function.aggregation**, vous allez programmer les méthodes suivantes :

public double dZdW(**int** u, **int** v)

public double dZdA(**int** u, **int** v)

public double dZdB(**int** u)

L'objectif de ces méthodes seront d'évaluer les dérivées partielles de l'agrégation de la somme pondérée

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

par rapport à différents paramètres :

$$\frac{\partial z_u^{(k)}}{\partial w_{uv}^{(k)}} = a_v^{(k-1)}$$

(Dérivée partielle par rapport au poids $w_{uv}^{(k)}$)

$$\frac{\partial z_u^{(k)}}{\partial a_v^{(k-1)}} = w_{uv}^{(k)}$$

(Dérivée partielle par rapport à l'entrée $a_v^{(k-1)}$)

$$\frac{\partial z_u^{(k)}}{\partial b_u^{(k)}} = 1$$

(Dérivée partielle par rapport au biais $b_u^{(k)}$)

Dans cette classe, vous pouvez avoir accès à $a_v^{(k-1)}$, $w_{uv}^{(k)}$ et $b_u^{(k)}$ grâce aux paramètres suivants :

- Le paramètre **this**.forward_inputs correspond à $a_v^{(k-1)}$ où *v* est l'indice dans le tableau.
- Le paramètre **this**.W donne accès à $w_{uv}^{(k)}$ à l'aide de l'appel W.getMatrix( ).
- Le paramètre **this**.B donne accès à $b_u^{(k)}$ à l'aide de l'appel B.getVector( ).

Présentement, ces méthodes retournent des exceptions de type **SNoImplementationException** spécifiant que ces méthodes n'ont pas été implémentées. Complétez l'implémentation de ces trois méthode et retournez les bonnes valeurs en utilisant judicieusement les indices u et v.

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

7.4 La dérivée de l'erreur quadratique

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 derivateQuadraticError(**double** calculated, **double** expected)

L'objectif de cette méthode sera d'évaluer la dérivée partielle de la fonction d'erreur quadratique par rapport à l'activation du réseau $a_u^{(L)}$ pour un seul neurone :

$$\frac{\partial C}{\partial a_u^{(L)}} = a_u^{(L)} - y_u \qquad \text{où} \qquad C = \frac{1}{2}\left(a_u^{(L)} - y_u\right)^2$$

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez l'implémentation et retournez $\partial C / \partial a_u^{(L)}$ à l'aide de $a_u^{(L)}$ correspondant à calculated et y_u correspondant à expected.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMath** du ***package* sim.nn.function** 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 7.4 :

Pourquoi la fonction d'erreur quadratique $C_{quad} = \frac{1}{2}\left(a_u^{(L)} - y_u\right)^2$ est mathématiquement plus intéressant qu'une fonction d'erreur linéaire de type $C_{lin} = a_u^{(L)} - y_u$?

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

public double dCdA(**int** u)

Ceci permettra à la fonction d'erreur energie (de classe **SEnergyErrorFunction**) de calculer la dérivée de l'erreur C pour le neurone d'indice u par rapport à $a_u^{(L)}$ d'indice u.

La propagation de l'erreur de la propagation du gradient

Pour réaliser la propagation du gradient, il faut d'une certaine façon activer le réseau dans le sens contraire (de la fin vers l'avant) afin de faire cheminer l'erreur finale entre $a_u^{(L)}$ et y_u générée par la fonction d'erreur $C = C\left(a^{(L)}, y\right)$ à l'ensemble des couches du réseau pour l'ensemble des neurones. Ainsi, il faut réaliser une progation de l'erreur (*back propagation*) de la fin du réseau vers le début du réseau.

Dans l'infrastructure de ce programme, nous avons les paramètres suivants pour toutes les fonctions :

forward_inputs : Entrée d'une fonction lors de l'activation (vers l'avant).

forward_outputs : Sortie d'une fonction lors de l'activation (vers l'avant).

backward_inputs : Entrée d'une fonction lors de la propagation de l'erreur (vers l'arrière). Ce terme correspond à l'erreur à propager dans la fonction.

backward_outputs : Sortie d'une fonction lors de la propagation de l'erreur (vers l'arrière). Ce terme correspond à l'erreur qui a été propagée par la fonction.

Dans les étapes qui vont suivrent, vous aurez à programmer des expressions de propagation de l'erreur avec le paramètre backward_inputs et vous devrez les sauvegarder dans le paramètre backward_outputs pour différentes fonctions d'erreur, d'activation et d'agrégation :

$$\Delta_u^{C(L)}, \quad \Delta_u^{a(k)}, \quad \text{et} \quad \Delta_v^{z(k)}$$

Par la suite, vous devrez programmer la propagation des erreurs dans une couche de neurones pour finalement l'appliquer à l'ensemble du réseau.

Toutes les équations qui seront à programmer seront basées sur l'expression de la différentielle de la fonction d'erreur $\mathrm{d}C = \mathrm{d}C\left(a^{(L)}, y\right)$ correspondant à

$$\mathrm{d}C = \sum_{i=0}^{N^{(L)}-1} \frac{\partial C}{\partial a_i^{(L)}} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \left(\begin{array}{l} \displaystyle\sum_{j=0}^{N^{(L-1)}-1} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} \mathrm{d}w_{ij}^{(L)} + \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} \mathrm{d}b_i^{(L)} \\ + \displaystyle\sum_{j=0}^{N^{(L-1)}-1} \frac{\partial z_i^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}} \left(\begin{array}{l} \displaystyle\sum_{k=0}^{N^{(L-1)}-1} \frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} \mathrm{d}w_{jk}^{(L-1)} + \frac{\partial z_j^{(L-1)}}{\partial b_j^{(L-1)}} \mathrm{d}b_j^{(L-1)} \\ + \displaystyle\sum_{k=0}^{N^{(L-2)}-1} \frac{\partial z_j^{(L-1)}}{\partial a_k^{(L-2)}} \mathrm{d}a_k^{(L-2)} \end{array} \right) \end{array} \right)$$

(Équation de la différentielle de la fonction d'erreur jusqu'à la couche de profoncteur $n = 2$)

Avec :

- La fonction d'erreur $C = C\left(a^{(L)}, y\right)$
- L'agrégation $z_u^{(k)} = z_u^{(k)}\left(a_v^{(k-1)}, w_{uv}^{(k)}, b_u^{(k)}\right)$
- L'activation $a_u^{(k)} = a_u^{(k)}\left(z_u^{(k)}\right)$
- L'indice de sortie $u \in \left[0 \ldots N^{(k)} - 1\right]$ du neurone (ligne dans la matrice des poids $w_{uv}^{(k)}$)
- L'indice d'entrée $v \in \left[0 \ldots N^{(k-1)} - 1\right]$ du neurone (colonne dans la matrice des poids $w_{uv}^{(k)}$)
- L'indice de la couche $k \in \left[0 \ldots L\right]$ d'un réseau comportant $L+1$ couches

8.1 La propagation de l'erreur de la fonction d'erreur *C*

Fichier à modifier : **SErrorFunction.java**

Prérequis : 7.4

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

protected double backward(**int** u)

Le but de cette méthode est d'évaluer la propagation de l'erreur d'un neurone u d'une fonction d'erreur $C = C\left(a^{(L)}, y\right)$ tel que

$$\Delta_u^{C(L)} = \frac{\partial C}{\partial a_u^{(L)}} \quad .$$

Pour ce faire, vous n'aurez qu'à retourner le résultat de la méthode dCdA(int u) pour le neurone u puisque l'erreur à propager sera « 1 » pondérée par $\partial C / \partial a_u^{(L)}$ pour le neurone u. Bien que cette dernière soit abstraite pour la classe **SErrorFunction**, elle sera définie pour toutes les fonctions d'erreur héritant de **SErrorFunction** (comme **SEnergyErrorFunction**).

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

8.2 La propagation de l'erreur de la fonction d'activation *A*

Fichier à modifier : **SActivationFunction.java**

Prérequis : 7.1

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

protected double backward(**int** u)

Le but de cette méthode est d'évaluer la propagation de l'erreur d'un neurone u d'une fonction d'activation[1] $a_u^{(k)} = a_u^{(k)}\left(z_u^{(k)}\right)$ tel que

$$\Delta_u^{a(k)} = \Delta_u \frac{\partial a_u^{(k)}}{\partial z_u^{(k)}} \qquad \text{où} \qquad \Delta_u^{a(k)} = \frac{\partial C}{\partial z_u^{(k)}} \quad \text{et} \quad \Delta_u = \frac{\partial C}{\partial a_u^{(k)}} \quad .$$

Pour ce faire, utilisez **this**.backward_inputs représentant l'erreur à propager Δ_u dans la fonction d'activation et la méthode dAdZ(int u) représentant la dérivée partielle de la fonction d'activation (abstrainte pour la classe **SActivationFunction**, mais bien définie pour les classe **SSigmoidFunction** et **STanhFunction**).

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

[1] Il est à noté que cette version de la propagation de l'erreur ne s'applique pas aux fonctions d'activation de type *full-connected* comme par exemple la fonction softmax.

8.3 La propagation de l'erreur de la fonction d'agrégation *Z*

Fichier à modifier : **SAggregationFunction.java**

Prérequis : 7.3

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

protected double backward(**int** v)

Le but de cette méthode est d'évaluer la propagation de l'erreur d'un neurone v d'une fonction d'agrégation

$z_u^{(k)} = z_u^{(k)}\left(a_v^{(k-1)}, w_{uv}^{(k)}, b_u^{(k)}\right)$ tel que $\Delta_v^{z(k)} = \sum_{u=0}^{N^{(k)}-1} \Delta_u \frac{\partial z_u^{(k)}}{\partial a_v^{(k-1)}}$ où $\Delta_v^{z(k)} = \frac{\partial C}{\partial a_v^{(k-1)}}$ et $\Delta_u = \frac{\partial C}{\partial z_u^{(k)}}$.

Pour ce faire, utilisez **this**.backward_inputs représentant l'erreur à propager Δ_u dans la fonction d'agrégation et la méthode dZdA(**int** u, **int** v) représentant la dérivée partielle de la fonction d'agrégation.

Exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SAgregationFunctionTest** du ***package*** **sim.nn.function.aggregation** 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 8.3 :

Dans l'équation de la propagation de l'erreur de la fonction d'agrégation $z_u^{(k)} = z_u^{(k)}\left(a_v^{(k-1)}, w_{uv}^{(k)}, b_u^{(k)}\right)$, pourquoi retrouve t-on une sommation ce qui n'a pas été le cas pour la fonction activation $a_u^{(k)} = a_u^{(k)}\left(z_u^{(k)}\right)$?

8.4 La propagation abstraite des erreurs sur plusieurs neurones

Fichier à modifier : **SAbstractNeuralFunction.java**

Prérequis : 8.1, 82 et 8.3

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

protected double[ ] backwardPass( )

Cette méthode a pour but de généraliser, pour n'importe quel type de fonction (agrégation, activation et erreur), le calcul de la propagation de l'erreur pour l'ensemble des neurones de la fonction.

Pour ce faire, vous devrez utiliser la méthode abstraite

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

disponible dans cette classe calculant la propagation de l'erreur pour le neurone u.

Pour réaliser votre implémentation, utilisez le champ[2] local de la classe

protected double[ ] backward_outputs

correspondant à la sortie de la fonction en mode vers l'arrière pour enregister vos calculs réalisés par la méthode backward(**int** u) pour chacun des u neurones. Lorsque la propagation de l'erreur pour vos neurones aura été calculée, retournez ce résultat (**return this**.backward_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**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

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

public double[ ] backPropagation(**double**[ ] delta) **throws** IllegalArgumentException

afin de permettre à toutes les fonctions neuronales de pouvoir réaliser la propagation des erreurs à partir de l'erreur (delta) à propager dans la fonction.

8.5 La propagation de l'erreur d'une couche de neurones

Fichier à modifier : **SFullConnectedLayer.java**

Prérequis : 8.4

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

public double[ ] backPropagation(**double**[ ] delta) **throws** IllegalArgumentException

Le but de cette méthode est d'évaluer la propagation de l'erreur d'une couche de neurones ayant une fonction d'agrégation Z et un fonction d'activation A.

Pour réaliser la propagation de l'erreur (propagation vers l'arrière), utilisez delta représentant l'erreur à propager dans vos fonctions Z et A. Utilisez la méthode

public double[ ] backPropagation(**double**[ ] delta) **throws** IllegalArgumentException

sur vos objets **this**.A et **this**.Z de votre classe pour réaliser vos calculs. N'oubliez pas que l'activation était dans l'ordre $Z \rightarrow A$. Ainsi, la propagation de l'erreur sera dans l'ordre $A \rightarrow Z$.

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.

[2] Par exemple, le champ backward_outputs correspond à l'erreur propagée $\Delta_u^{z(k)}$ par la fonction d'agrégation $z_u^{(k)}$ dans la classe **SAggregationFunction**.

8.6 Retour sur le prélaboratoire, partie 2 (propagation des erreurs)

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : aucun

Prérequis : 8.5

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 2 (propagation de l'erreur) »** 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	Erreur Δ
$W = \begin{pmatrix} 1.0 & 4.0 \\ 3.0 & 2.0 \\ -2.0 & -3.0 \end{pmatrix}$	$b = \begin{pmatrix} 0.8 \\ 0.5 \\ -0.2 \end{pmatrix}$	$x = \begin{pmatrix} -0.4 \\ 0.6 \end{pmatrix}$	$\Delta = \begin{pmatrix} 0.2 \\ 0.1 \\ -0.1 \end{pmatrix}$

Répondez aux questions suivantes dans le cahier de réponse :

Question 8.6.1 :

Quel est le résultat de l'erreur propagée de votre réseau ?

Question 8.6.2 :

Est-ce que la l'erreur propagée 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 prelaboratoirePartie2( )
```

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

8.7 La propagation de l'erreur d'un réseau

Fichier à modifier : **SNetwork.java**

Prérequis : 8.5

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

```
public double[ ] backPropagation(double[ ] delta) throws IllegalArgumentException
```

Le but de cette méthode est d'évaluer la propagation de l'erreur d'un réseau de plusieurs couches.

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 la propagation de l'erreur (propagation de l'erreur vers l'arrière), utilisez delta représentant l'erreur à propager depuis la dernière couche du réseau (en indice *L*) jusqu'à la 1re couche du réseau (en indice 0). Lorsque vous aurez évaluez l'erreur propagée par la couche d'entrée du réseau, retournez ce résultat.

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.

Le gradient de la fonction d'erreur

Afin d'évaluer le gradient de la fonction d'erreur ∇C à partir d'une fonction d'erreur

$$C = C\left(a^{(L)}, y\right)$$

où l'activation du réseau $a^{(L)}$ dépend d'un vecteur d'entrée $a^{(*)}$ et de l'ensemble des poids $w_{uv}^{(k)}$ et biais $b_u^{(k)}$ du réseau, nous avons effectué la différentielle de cette fonction d'erreur pour obtenir l'expression suivante où les termes $\Delta_u^{a(k)}$, et $\Delta_v^{z(k)}$ ont été introduit pour simplifier l'expression :

$$\mathrm{d}C = \left(\begin{array}{l} \displaystyle\sum_{i=0}^{N^{(L)}-1} \sum_{j=0}^{N^{(L-1)}-1} \Delta_i^{a(L)} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} \mathrm{d}w_{ij}^{(L)} + \sum_{i=0}^{N^{(L)}-1} \Delta_i^{a(L)} \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} \mathrm{d}b_i^{(L)} \\ + \left(\begin{array}{l} \displaystyle\sum_{j=0}^{N^{(L-1)}-1} \sum_{k=0}^{N^{(L-1)}-1} \Delta_j^{a(L-1)} \frac{\partial z_j^{(L-1)}}{\partial w_{jk}^{(L-1)}} \mathrm{d}w_{jk}^{(L-1)} + \sum_{j=0}^{N^{(L-1)}-1} \Delta_j^{a(L-1)} \frac{\partial z_j^{(L-1)}}{\partial b_j^{(L-1)}} \mathrm{d}b_j^{(L-1)} \\ + \displaystyle\sum_{k=0}^{N^{(L-2)}-1} \Delta_k^{z(L-1)} \mathrm{d}a_k^{(L-2)} \end{array} \right) \end{array} \right)$$

(Équation de la différentielle de la fonction d'erreur avec propagation d'erreur jusqu'à la couche de profoncteur $n = 2$)

En effectuant l'ensemble des sommations de l'équation, nous obtenons les composantes du gradient

$$\frac{\partial C}{\partial w_{uv}^{(k)}} = \Delta_u^{a(k)} \frac{\partial z_u^{(k)}}{\partial w_{uv}^{(k)}} \qquad \text{et} \qquad \frac{\partial C}{\partial b_u^{(k)}} = \Delta_u^{a(k)} \frac{\partial z_u^{(k)}}{\partial b_u^{(k)}}$$

pour tous les paramètres de toutes les couches du réseau.

Mathématiquement, le gradient de la fonction d'erreur ∇C correspond au vecteur suivant :

$$\nabla C = \left(\frac{\partial C}{\partial w_{yz}^{(0)}}, \frac{\partial C}{\partial b_y^{(0)}}, \frac{\partial C}{\partial w_{xy}^{(1)}}, \frac{\partial C}{\partial b_x^{(1)}}, \cdots, \frac{\partial C}{\partial w_{uv}^{(k)}}, \frac{\partial C}{\partial b_u^{(k)}}, \cdots, \frac{\partial C}{\partial w_{jk}^{(L-1)}}, \frac{\partial C}{\partial b_j^{(L-1)}}, \frac{\partial C}{\partial w_{ij}^{(L)}}, \frac{\partial C}{\partial b_i^{(L)}} \right)$$

Ce vecteur donnera la direction opposée dans laquelle il faudra faire varier les poids $w_{uv}^{(k)}$ et $b_u^{(k)}$ pour que l'erreur occasionnée par la différence entre $a^{(L)}$ et y diminue. Si l'erreur diminue après la modification des poids et biais, on pourra affirmer que « le réseau a réalisé un apprentissage » à reproduire l'activation désirée.

9.1 La dérivée partielle de la fonction d'erreur *C* par rapport au biais *b*

Fichier à modifier : **SAggregationFunction.java**

Prérequis : 7.5

Dans la classe **SAggregationFunction** disponible dans le ***package*** **sim.nn.function.aggregation**, vous allez programmer les méthodes suivantes :

public double dCdB(**int** u)

Les buts de ces deux méthodes est d'évaluer les dérivées partielles de la fonction d'erreur par rapport au biais $b_u^{(k)}$:

$$\frac{\partial C}{\partial b_u^{(k)}} = \Delta_u^{a(k)} \frac{\partial z_u^{(k)}}{\partial b_u^{(k)}}$$

(Dérivée partielle par rapport au biais $b_u^{(k)}$)

Pour ce faire, utilisez **this**.backward_input représentant l'erreur $\Delta_u^{a(k)}$ dans la classe **SAggregationFunction** et la méthodes dZdB(**int** u) représentant la dérivée partielle de la fonction d'agrégation par rapport au biais.

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

(FACULATIF) Cette fonctionnalité a été introduite pour vous dans la classe **SFullConnectedGradientLayer** disponible dans le ***package*** **sim.nn.gradient** pour enregistrer les expressions de $\partial C / \partial b_u^{(k)}$ pour une couche du réseau et dans la classe **SErrorFunctionGradient** disponible dans le ***package*** **sim.nn.gradient** pour représenter le vecteur gradient

$$\nabla C = \left(\frac{\partial C}{\partial w_{yz}^{(0)}}, \frac{\partial C}{\partial b_y^{(0)}}, \frac{\partial C}{\partial w_{xy}^{(1)}}, \frac{\partial C}{\partial b_x^{(1)}}, \cdots, \frac{\partial C}{\partial w_{uv}^{(k)}}, \frac{\partial C}{\partial b_u^{(k)}}, \cdots, \frac{\partial C}{\partial w_{jk}^{(L-1)}}, \frac{\partial C}{\partial b_j^{(L-1)}}, \frac{\partial C}{\partial w_{ij}^{(L)}}, \frac{\partial C}{\partial b_i^{(L)}} \right) .$$

9.2 La dérivée partielle de la fonction d'erreur *C* par rapport au poids *w*

Fichier à modifier : **SWeigthedSumFunction.java**

Prérequis : 7.5

Dans la classe **SWeightedSunFunction** disponible dans le ***package*** **sim.nn.function.aggregation**, vous allez programmer les méthodes suivantes :

public double dCdW(**int** v, **int** u)

Les buts de ces deux méthodes est d'évaluer les dérivées partielles de la fonction d'erreur par rapport au poids $w_{uv}^{(k)}$:

$$\frac{\partial C}{\partial w_{uv}^{(k)}} = \Delta_u^{a(k)} \frac{\partial z_u^{(k)}}{\partial w_{uv}^{(k)}}$$

(Dérivée partielle par rapport au poids $w_{uv}^{(k)}$)

Pour ce faire, utilisez **this**.backward_input représentant l'erreur $\Delta_u^{a(k)}$ dans la classe **SWeightedSumFunction** et les méthodes dZdW(**int** u, **int** v) représentant la dérivée partielle de la fonction d'agrégation par rapport au poids.

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

(FACULATIF) Cette fonctionnalité a été introduite pour vous dans la classe **SFullConnectedGradientLayer** disponible dans le ***package*** **sim.nn.gradient** pour enregistrer les expressions de $\partial C / \partial w_{uv}^{(k)}$ pour une couche du réseau et dans la classe **SErrorFunctionGradient** disponible dans le ***package*** **sim.nn.gradient** pour représenter le vecteur gradient

$$\nabla C = \left(\frac{\partial C}{\partial w_{yz}^{(0)}}, \frac{\partial C}{\partial b_{y}^{(0)}}, \frac{\partial C}{\partial w_{xy}^{(1)}}, \frac{\partial C}{\partial b_{x}^{(1)}}, \cdots, \frac{\partial C}{\partial w_{uv}^{(k)}}, \frac{\partial C}{\partial b_{u}^{(k)}}, \cdots, \frac{\partial C}{\partial w_{jk}^{(L-1)}}, \frac{\partial C}{\partial b_{j}^{(L-1)}}, \frac{\partial C}{\partial w_{ij}^{(L)}}, \frac{\partial C}{\partial b_{i}^{(L)}} \right).$$

9.3 Le calcul du gradient de la fonction d'erreur

Fichier à modifier : **SGradientAlgorithm.java**

Prérequis : 9.1 et 9.2

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

```
public static SErrorFunctionGradient computeGradient(SNetwork network, SErrorFunction error_function,
                                                     SErrorFunctionGradient gradient, SNNData data)
```

Cette méthode a pour but d'alimenter le gradient de la fonction d'erreur gradient à l'aide de la propagation de l'erreur de la fonction d'erreur error_function dans le réseau network provenant de l'activation de la donnée data.

Pour réaliser cette tâche, 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 propagation de l'erreur de la fonction d'erreur error_function à l'aide de l'instruction error_function.backPropagation( ). N'oubliez pas de sauvegarder cette erreur propagée dans un tableau de type ***double*** (double[ ]).
4. Effectuer la propagation de l'erreur dans le réseau network à partir de l'erreur propagée par la fonction d'erreur (calculé en 3). L'erreur à l'entrée du réseau n'aura pas besoin d'être sauvegarder.
5. Évaluer le gradient de la fonction d'erreur avec la contribution du réseau à l'aide de l'instruction gradient.increase(network). Il est très important d'utiliser la méthode « increase » et non la méthode « add », car elle ne sont pas équivalente (lire la documentation au besoin). C'est dans cette méthode que les calculs de $\partial C / \partial W_{uv}^{(k)}$ et $\partial C / \partial b_{u}^{(k)}$ seront effectués et enregistrés.
6. Retourner le paramètre gradient par l'instruction **return** gradient.

Exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGradientAlgorithmTest** du ***package*** **sim.nn.gradient** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

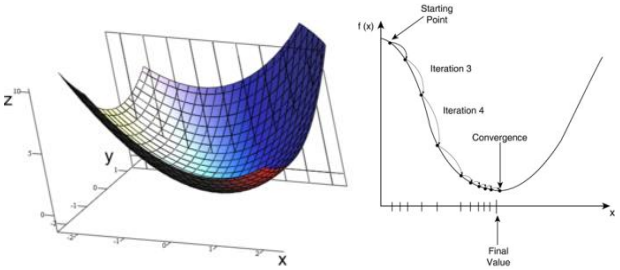
Répondez à la question suivante dans le cahier de réponse :

Question 9.3 :

Quel est le problème principal des valeurs du gradient de la fonction d'erreur lorsqu'on applique cette technique à un réseau contenant beaucoup de couches ?

L'apprentissage du réseau

L'apprentissage d'un réseau consiste à modifier les poids (*weights*) et biais (*bias*) de toutes les couches de neurones dans la direction opposée au gradient de la fonction d'erreur obtenu par la descente du gradient. En modifiant les paramètres du réseau de la sorte, on réduit l'erreur de celui-ci en lien avec le vecteur d'entraînement utilisé lors de la descente. Si ce processus est suffisamment répété sur un échantillon assez grand et varié de vecteurs d'entraînement, le réseau pourra bien représenter la fonction mathématique désirée.



https://www.quora.com/Does-Gradient-Descent-Algo-always-converge-to-the-global-minimum

Illustration d'une descente du gradient avec mise à jour d'une fonction afin de minimiser sa valeur.

Dans cette section du laboratoire, vous devrez implémenter l'algorithme de l'apprentissage en lien avec le taux d'apprentissage (*learning rate*) et l'appliquer au poids et biais d'un réseau.

10.1 L'apprentissage

Fichier à modifier : **SNeuralMath.java**

Prérequis : aucun

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

```
public static double learning(double V, double gradient, double alpha)
```

Le rôle de cette méthode sera de permettre la mise à jour les paramètres $w_{uv}^{(k)}$ et $b_u^{(k)}$ de votre réseau correspondant ainsi à l'apprentissage. Vous utiliserez cette méthode dans le but de réaliser les calculs suivants :

$$\tilde{w}_{uv}^{(k)} = w_{uv}^{(k)} - \alpha \frac{\partial C}{\partial w_{uv}^{(k)}}$$

(mise à jour d'un poids)

et

$$\tilde{b}_u^{(k)} = b_u^{(k)} - \alpha \frac{\partial C}{\partial b_u^{(k)}}$$

(mise à jour d'un biais)

Dans ces deux équations, on remarque la présence d'un signe négatif. Cela signifie que l'on veut modifier nos valeurs $w_{uv}^{(k)}$ et $b_u^{(k)}$ dans le sens opposé au gradient ce qui aura pour but de réduire l'erreur, car le gradient point dans la direction où il y aura augmentation de l'erreur.

Afin de réduire la programmation à réaliser, vous généralisez ces deux équations[3] par l'expression

$$\tilde{v} = v - \alpha \frac{\partial C}{\partial v}$$

(mise à jour du paramètre *v*)

où v correspond au paramètre V, $\partial C / \partial v$ correspond au paramètre gradient et α correspond au paramètre alpha.

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Complétez l'implémentation afin d'obtenir une valeur modifiée par l'apprentissage.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SNeuralMath** 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 10.1.1 :

Pourquoi un terme négatif a été introduit dans « l'équation de l'apprentissage » ?

Question 10.1.2 :

Quel problème peut survenir lors de la phase de l'apprentissage si des erreurs ont été propagées par des fonctions d'activation où la pente était près de la valeur zéro ?

(FACULATIF) Cette fonctionnalité a été introduite pour vous dans les deux classes **SBias** et **SWeights** disponibles dans le ***package*** **sim.nn.parameter** dans la méthode abstraite

protected void learn(SFullConnectedGradientLayer gradient, **double** learning_rate)

afin de permettre de modifier les poids (*weights*) et les biais (*bias*) d'une couche de neurones dans la phase d'apprentissage. Ainsi, nous pourrons utiliser la méthode learn(...) sur un réseau afin que l'apprentissage puisse se réaliser couche par couche (déjà programmé pour vous) avec l'usage du gradient de la fonction d'erreur.

[3] Dans un réseau de neurones dont l'activation se fait par batch (*batch-activation*), il existe une fonction de normalisation (*batch-normalization*) qui utilise deux autres paramètres : *scale* et *shift*.

L'entrainement d'un réseau

Un réseau de neurones entraîné est composé de couches de neurones dont les valeurs des poids et biais ont été fixées avec des valeurs réelles judicieusement déterminées. En réalisant l'activation de ce type de réseau avec un vecteur d'entrée quelconque, le résultat obtenu correspond habituellement à un résultat attendu. La qualité du résultat varie selon plusieurs facteurs : la quantité d'entraînement, la qualité des vecteurs d'entraînement, l'architecture du réseau, etc.

Dans cette partie du laboratoire, vous allez entraîner un réseau de neurones avec différentes collections de données et analyser la qualité des activations générés par le réseau.

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

Un réseau de neurones entraîné permet d'établir des liens entre les différents paramètres d'un vecteur d'entrée ce qui correspond à réaliser un apprentissage.

Dans un protocole d'entraînement de base[4], vous retrouverez les paramètres suivants :

- Données d'entraînement (*data*) :

  Nombre de données utilisées pour améliorer le réseau. Dans un entraînement de base, on utilise une seule donnée à la fois pour réaliser le calcul de la propagation du grandient.

- Époque (*epoch*) :

  Nombre de fois que l'ensemble des données d'entraînement ont été utilisées pour entraîner le réseau.

- Entraînement (*training*) :

  Nombre de fois qu'il y a eu modification des paramètres du réseau. Cet étape correspond à un pas de la descente du gradient de la fonction d'errreur.

- Iteration (*iteration*) :

  Bloc d'entraînement à réaliser avant d'évaluer le taux de classification du réseau.

- Cycle (*cycle*) :

  Nombre de bloc d'entraînement à réaliser avant de compléter l'entraînement du réseau.

- Taux d'apprentissage (*learning rate*)

  Facteur multiplicatif correspondant à un taux de modification des paramètres du réseau. Ce facteur est utilisé dans la phase de l'apprentissage du réseau.

[4] Un protocole avancé utilisera le *mini-batch* et le *batch-activation* pour évaluer le gradient moyen de la fonction d'erreur à chaque étape de la descente du gradient. De plus, on peut réaliser la descente du gradient en utilisant un gradient avec moment (avec inertie).

11.1 Le protocole d'entraînement

Fichier à modifier : **STrainingAlgorithm.java**

Prérequis : 9.4, 10.2 et 10.3

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

```
public static void training(SNetwork network, SErrorFunction error_function, SErrorFunctionGradient gradient,
                            SNNData data, double learning_rate)
```

L'objectif de cette méthode sera de réaliser l'entraînement du réseau à l'aide d'une donnée.

Pour ce faire, vous devrez :

1. Débuter par vider le gradient de la fonction d'erreur gradient avec l'instruction gradient.clear( ) afin de vous assurer que celui-ci est bien vide avant de l'utiliser dans vos calculs.
2. Effectuer le calcul le gradient de la fonction d'erreur avec la méthode

   ```
   public static SErrorFunctionGradient computeGradient(SNetwork network, SErrorFunction error_function,
                                                         SErrorFunctionGradient gradient, SNNData data)
   ```

   disponible dans la classe **SGradientAlgorithm** que vous avez programme précédemment.
3. Réaliser l'apprentissage du réseau (network) en utilisant la méthode learn(…) du réseau avec le gradient (gradient) et le taux d'apprentissage (learning_rate).

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.

Répondez à la question suivante dans le cahier de réponse :

Question 11.1 :

Typiquement, un réseau non calibré est initialisé avec des valeurs aléatoires entre – 1 et + 1 pour l'ensemble des poids et biais que le constitue. Pourquoi ne pas tout simplement intialiser le tout avec des valeurs égales à zéro ?

11.2 L'apprentissage de Breast Cancer Wisconsin (Diagnostic) Data Set

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : **SIMLearning.java**

Prérequis : 11.1

Dans la classe **SIMLearning** disponible dans le ***package*** **sim.application**, vous allez réaliser l'apprentissage de la base de données Breast Cancer Wisconsin Data Set avec un réseau **10 x 1**.

Dans la méthode

```
private static void learningWDBC( )
```

de la classe **SIMLearning**, vous aurez le loisir de modifier les deux constantes

```
final int CYCLE;
final double LEARNING_RATE;
```

Dans le protocole d'apprentissage par défaut, il y aura **50 cycles** d'entraînement (CYCLE = 50) avec une itération par cycle utilisant une donnée choisie aléatoirement dans la collection des vecteurs d'entraînement. Le **taux d'apprentissage** sera de **0.1** (LEARNING_RATE = 0.1). L'entraînement complet aura visité **50 données**.

Exécutez le programme **SIMLearning** et choisir l'option *(7) Apprentissage Breast Cancer Wisconsin (Diagnostic) Data Set*. À l'aide de l'affichage console, répondez aux questions ci-dessous. Modifier vos paramètres et relancez l'exécution afin de varier les scénarios d'apprentissage pour répondre à toutes les questions.

Répondez aux questions suivantes dans le cahier de réponse :

Question 11.2.1 :

Avec le protocole d'entraînement tel que CYCLE = 50 et LEARNING_RATE = 0.1, est ce que vous avez suffisamment de vecteurs d'entraînement pour obtenir un taux de classification raisonnable ? Justifiez votre réponse.

Question 11.2.2 :

Est-il possible d'atteindre un taux de classification de plus de 95% ? Si oui, combien de vecteur d'entraînement ont été nécessaire et à quel taux d'apprentissage ?

Question 11.2.3 :

Pour obtenir un taux de classification supérieur à 95%, est-il nécessaire de réaliser un entraînement sur l'ensemble des vecteurs d'entraînement (683 vecteurs disponibles) ? Justifiez votre réponse.

Question 11.2.4 :

Est-il possible d'utiliser moins de 50 vecteurs d'entraînement et obtenir un taux de classification de plus de 90 % ? Si oui, quel taux d'apprentissage (LEARNING_RATE) avez-vous utilisé ?

11.3 L'apprentissage de MNIST Data Set

Fichier à exécuter : **SIMLearning.java**

Fichier à modifier : **SIMLearning.java**

Prérequis : 11.1

Dans la classe **SIMLearning** disponible dans le ***package* sim.application**, vous allez réaliser l'apprentissage de la base de données MNIST Data Set avec un réseau **784 x 16 x 16 x 10**. Dans la méthode

```
private static void learningMNIST( )
```

de la classe **SIMLearning**, vous aurez le loisir de modifier les trois constantes

```
final int CYCLE;
final int ITERATION;
final double LEARNING_RATE;
```

afin de réaliser l'apprentissage des données MNIST sous différents scénarios d'apprentissage.

Dans le protocole d'apprentissage par défaut, il y aura **100 cycles** (CYCLE = 100) avec **100 itérations** (ITERATION = 100) par cycle utilisant une donnée choisie aléatoirement dans la collection des vecteurs d'entraînement. Le **taux d'apprentissage** sera de de **0.1** (LEARNING_RATE = 0.1). L'entrainement complet aura visité **10 000 données**.

Avant d'exécuter le programme, vous devrez installer la collection d'images (ce qui aura été réalisé en étape **6.4**).

Exécutez le programme **SIMLearning** et choisir l'option *(8) Apprentissage MNIST Data Set*. À l'aide de l'affichage console, répondez aux questions ci-dessous.

Modifier vos paramètres et relancez l'exécution afin de varier les scénarios d'apprentissage pour répondre à toutes les questions.

Question 11.3.1 :

Avec le protocole d'entraînement tel que CYCLE = 100, ITERATION = 100 et LEARNING_RATE = 0.1, est ce que vous avez suffisamment de vecteurs d'entraînement (10 000 vecteurs) pour obtenir un taux de classification raisonnable ? Justifiez votre réponse.

Question 11.3.2 :

Selon vous, avec un protocole d'entraînement tel que LEARNING_RATE = 0.1,

a) Quel est le meilleur taux de classification que le réseau 784 x 16 x 16 x 10 peut obtenir ? Modifiez au besoin les paramètres CYCLE et ITERATION pour faire varier votre protocole d'entraînement.
b) Combiens de vecteur d'entraînement ont été nécessaire pour obtenir ce meilleur taux de classification ?

Question 11.3.3 :

Avec le protocole d'entraînement CYCLE = 100, ITERATION = 100, et LEARNING_RATE = 0.9, le taux de classification du réseau évolue tout au long de l'entraînement de façon particulier. Identifiez

a) Une caractéristique positive.
b) Une caractéristique négative.

Conclusion

Félicitations ! Vous avez compléter l'implémentation de l'apprentissage de votre réseau de neurones et vous êtes apte à l'entraîner 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.