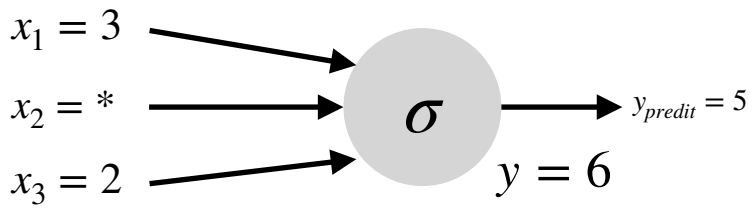
Projet INDIVIDUEL de Modélisation et Programmation Orientée Objet à rendre pour le 01 juin 2021 avant minuit (sources et rapport)

On veut écrire une calculatrice simple fondée sur l'Intelligence Artificielle. **L'objectif** de cette application **n'est pas l'IA** mais la modélisation et la programmation orientées objet en Java.

Objectif

L'objectif est donc de calculer la somme ou la multiplication de deux chiffres compris entre 0 et 10. Vous pourrez tester de plus grandes valeurs en modifiant la taille des entrées/sorties du système. L'IA prendra en entrée de son modèle (ici un neurone simple σ) un vecteur correspondant à l'opération x a réalisé. Il faudra coder l'opérande (x_2 ici) avec une valeur numérique (1 pour + et 0 pour la multiplication *).



Exemple de calcul

L'exemple illustré dans la figure ci-dessus comporte une entrée $x = [3,0,2]$ et une sortie $y = [0,0,0,0,0,0,1,0,\ldots,0]$ avec un 1 à la 7ème position correspondant à la valeur 6.

L'IA estimera depuis l'entrée x un résultat y_{predit} lors de la phase de pass avant (forward) à l'aide des paramètres du modèles (les poids w et le biais b ainsi que la fonction de transfert σ). Pour ajuster les paramètres du modèle dans l'objectif d'améliorer la qualité de la prédiction, l'IA corrige lors de la passe arrière (backward) sa manière de prédire la sortie à l'aide de l'erreur (loss) observée entre le résultat prédit y_{predit} et le bon résultat y.

Dans le cas de l'exemple, $y_{predit} = [0,0,0,0,0,1,0,\ldots,0]$ (vous aurez surtout des valeurs proches de 0 ou proches de 1 comme 0.001 et 0.89 dans y_{predit} en lieu est place de 0 et de 1) et l'erreur sera alors $y - y_{predit} = [0,0,0,0,0,-1,1,0\ldots,0]$.

Ce projet développera une IA appelée *MultiLayer Perceptron* (MLP) permettant de réaliser des calculs simples d'addition et de multiplication. La prochaine section vous donnera les informations sur le fonctionnement d'un perceptron ou neurone puis du MLP. Ensuite une section est dédiée aux algorithmes (à arranger au besoin) des passes avant et arrière de chacun des éléments composant le MLP.

I) MultiLayer Perceptron

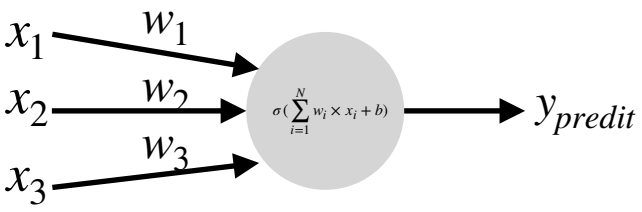
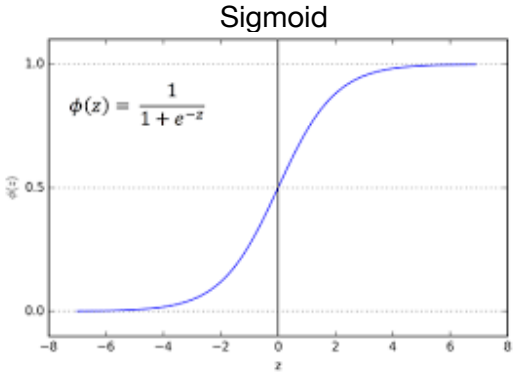


Figure 1

La figure 1 présente un perceptron ou neurone recevant en entrée un tableau de double $x = [x_1, x_2, x_3]$ et délivrant en sortie un double (y_{predit}) lors de la phase de passe avant (forward) . Cette sortie est déterminée en multipliant les entrées avec les poids $w = [w_1, w_2, w_3]$ (tableau de double) puis en sommant le résultat avec un bias b (double). Une fonction de transfert σ est ensuite appliquée au résultat pour obtenir y_{predit}. Lors de la passe arrière (backward), y_{predit} est comparée à la bonne sortie y pour en déterminer l'erreur (loss $= y_{predit} - y$) et corriger le modèle pour améliorer la qualité de la prédiction.

σ Fonctions de transfert (Sigmoid ou tanh) :



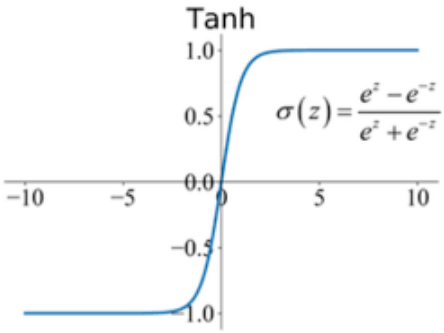


Figure 2

Le MultiLayer Perceptron ou MLP présenté Figure 3 comporte une couche (layer) l_1 contenant deux neurones (σ_1, σ_2) recevant en entrée $x = [x_1, x_2, x_3]$ et délivrant en sortie deux doubles h_1 et h_2. Une seconde couche l_2 compose le MLP contenant un seul neurone σ_y et fournissant en sortie du MLP la prédiction y_{predit}.

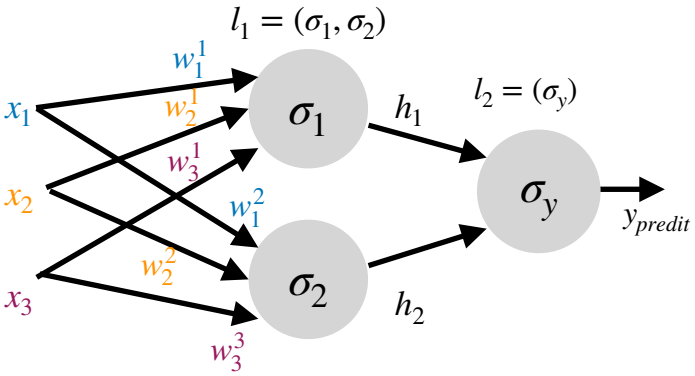


Figure 3

II) Algorithmes de passes avant et arrière des éléments du MLP

Algorithmes de passe avant (forward) :

MLP :

```
for (int i = 0; i < layers.length; i++)
      input = layers[i].forward(input);   //Appel la méthode forward de layer
return input ;
```

Layer :

```
double[] out = new double[this.size()];
int pos = 0 ;
for ( Neuron neuron )                     //Boucle for à modifier …
      out[pos++] = neuron.forward(input); //Appel la méthode forward de neurone
return out ;
```

Neurone :

```
this.xt = input ;
double output = 0.0 ;
for (int i = 0; i < input.length; i++)
      output += this.w[i] * input[i]  ;    //∑_{i=1}^{N} w_i × x_i (Figure 1)

this.yt = this.tf.ft(output + this.b);     //σ(∑_{i=1}^{N} w_i × x_i + b) (Figure 1)

return this.yt ;
```

Algorithmes de passe arrière (backward) :

MLP :

```
double[] loss = this.lossFunction.loss(output, predicted) ;
double[] dy = loss ;
for (int i = layers.length - 1 ; i >= 0; i -- )
      dy = layers[i].backward(dy);        //Appel la méthode backward de layer
return getLoss(loss) ;
```

Layer :

```
int pos = 0 ;
double[] dxt = new double[this.get(0).getWSize()];
for(INeuron neuron)
      dxt =neuron.backward(dy[pos++],dxt);//Appel la méthode backward de neurone
return dxt ;
```

Neurone :

```
double dxty = this.tf.dft(this.yt) * dy;
for (int i = 0; i < w.length; i++) {
      dxt[i] += this.w[i] * dxty ;
      this.w[i] += this.xt[i] * dxty * this.lr ;
}
this.b += dy * this.lr ;
return dxt ;
```

III) Classes à implémenter

A – Classes initiales

1) Ecrire une classe abstraite (ou autre chose …) `ITransfertFunction` comportant deux méthodes abstraites `ft` et `dft` permettant de calculer la sortie du neurone à l'aide dune fonction de transfert σ pour la phase de passe avant (forward avec `ft`) et d'une autre fonction nécessaire lors du calcul de la passe arrière (backward avec `dft`) (voire Figure 2). Ces deux méthodes prendront un argument de type double et un double pour retourner le résultat.

2) Ajouter deux classe dérivant de la classe `ITransfertFunction` permettant d'obtenir les fonctions de transfert dans le cas d'une sigmoid (`TransfertFunctionSigmoid`) et d'une tanh (`TransfertFunctionTanh`). Surdéfinissez pour ces deux classes les deux méthodes `ft` et `dft` contenues dans la classe `TransfertFunctionTanh` :

Sigmoid :

```
public double ft(double v) {return (1 / (1 + Math.exp(-v))) ;}
public double dft(double v) {return (v * ( 1 - v ) ) ;}
```

tanh :

```
public double ft(double v) {
      double e = Math.exp(v) ;
      double me = Math.exp(-v) ;
      return ( (e - me) / (e + me) ) ;
}
public double dft(double v) {return ( 1 - v * v ) ;}
```

3) Définir la classe abstraite (ou autre chose …) `ILoss` comportant une méthode abstraites `loss` de paramètres deux tableaux de double correspondant respectivement à la sortie désirée (y) et la sortie prédite (y_{predit}). Cette méthode `loss` retournera un tableau de double. Ecrire ensuite la class `LossDifference` dérivant de `ILoss` et redéfinir la méthode `loss` de `ILoss` qui retournera un tableau de double contenant la différence entre les deux arguments de la méthode y et y_{predit} sur chacun des indices.

4) De la même manière, définissez les classes abstraites (ou autre chose …) `IInitialiseWeights` et `IInitialiseBias` comportant respectivement chacune la méthode abstraite :

- `public double[] initWeights(int size);` pour `IInitialiseWeights`,
- `public double initBias();` pour `IInitialiseBias`.

Ajoutez deux classes héritant de `IInitialiseWeights` et `IInitialiseBias` appelées respectivement `InitialiseWeightsNormal` et `InitialiseBiasNormal` redéfinissant les méthodes `initWeights` et `initBias`. Celles-ci retournent, à l'aide de la méthode `Math.random()`, un tableau de taille `size` contenant des valeurs aléatoires comprises entre 0 et 1 pour `initWeights` et une valeur aléatoire comprise entre 0 et 1 pour `initBias`.

B – Classes du MLP

1) Ecrire une classe abstraite (ou autre …) `INeuron` qui contient deux méthodes abstraites : la méthode `forward` qui retourne un `double` représentant la sortie du neurone et prend en paramètre un tableau de double contenant les valeurs d'entrée; une méthode `backward` retournant l'accumulation du gradient de l'entrée (tableau de `double`) et prenant en paramètre ce gradient (tableau de `double`). Vous ajouterez à cette classe une méthode `getWSize` retournant la taille de `w`.

2) Ecrire une classe `NeuronLinear` dérivant de la classe `INeuron` contenant les attributs :
- w (`private double[] w`) représentant les poids associés à ce neurone,
- `b` de type `double` pour le biais,
- `lr` (`double`) pour le taux d'apprentissage,
- `xt` (`double[]`) sauvegardant l'entrée reçue par la méthode `forward` et `yt` sauvegardant la sortie générée `y` (`double`) par `forward`,
- `tf` de type `ITransfertFunction` nécessaire dans les algorithmes des méthodes `forward` et `backward` de `NeuronLinear`.

Redéfinissez également les deux méthodes `forward` et `backward` définies dans `INeuron` dont vous trouverez le pseudo-code dans la présentation du projet en début de sujet (section II).
Ajoutez la méthode : `public int getWSize() {return w.length ;}` permettant de récupérer la taille de w ainsi qu'un constructeur à 6 arguments permettant d'initialiser `NeuronLinear` : `public NeuronLinear(double lr, int inputSize, IInitialiseWeights initW, IInitialiseBias initB, ITransfertFunction tf) {…}`. Ce constructeur initialisera w et b à l'aide des méthode `initWeights` (avec le paramètre `inputSize`) et `initBias` des classes `InitWeightsNormal` et `InitBiasNormal` respectivement.

3) Ajoutez une classe `ILayer` abstraite (ou autres …) comportant deux méthodes abstraites retournant chacune un tableau de double :
- `forward` à un seul argument contenant l'entrée appelé `input` (tableau de double),
- `backward` avec un seul argument de contenant l'entrée appelé `dy` (tableau de double).

Cette classe sera une liste de la classe `java.utils` d'élément de type `INeuron`. Ainsi, les classes dérivant de `ILayer` pourront employer tout type de liste (`ArrayList`, `LinkedList`, etc.) et de neurone dérivant de `INeuron`.

4) Ecrire la classe `LayerLinear` dérivant de `ILayer` pour laquelle vous redéfinirez les deux méthodes `forward` et `backward` en vous aidant des algorithmes de la section II). Ecrire un constructeur permettant d'ajouter les `n` neurones (`NeuronLinear`) dans la liste : `public LayerLinear(int inputSize, int outputSize, double lr, IInitialiseWeights initWeights, IInitialiseBias initBias, ITransfertFunction tf){…}`.

5) Ecrire la classe abstraite (ou autres …) `IModel` contenant uniquement les méthodes abstraites `forward`, `backward` et `learn` :
- `public double[] forward(double[] input) ;`
- `public double backward(double[] output, double[] predicted) ;`
- `public void learn() ;`

Ajoutez la classe `MLP` dérivant de `IModel` et redéfinir les méthodes `forward` et `backward` à l'aide des algorithmes fournis dans la section II). La méthode `learn` combinera les deux passes avant et arrières nécessaires à l'apprentissage du MLP et sauvegardera l'erreur observée (attribut `loss`). MLP contiendra également les attributs :
- `private LayerLinear[] layers ;`
- `private ILoss lossFunction ;`
- `public double loss ;`
- `public double[] input ;`
- `public double[] output ;`
- `public double[] predicted ;`

Ainsi que les méthodes (à compléter au besoin) permettant de :
- Calculer l'erreur pour l'afficher dans la console :

```
public double getLoss(double[] dy) {
      double loss = 0.0 ;
      for (int i = 0; i < dy.length; i++)
            loss += dy[i];
      return loss /= dy.length ;
}
```

- Récupérer le plus grand indice contenu dans le tableau pour trouver la valeur prédite :

```
public static int getMaxIndice(double[] t) {
	int max = -1 ;
	double maxValue = -Double.MIN_VALUE ;
	for (int i = 0; i < t.length; i++) {
		if ( t[i] > maxValue ) {
			maxValue = t[i];
			max = i ;
		}
	}
	return max ;
}
```

C – Erreurs et Gestion des Exceptions

Les différentes phases d'apprentissage (forward et backward) du MLP et de calcul (forward) peuvent rencontrer des problèmes sur les différents éléments composant le MLP (neurone, layer, model ou loss).

1) Vous allez donc dans un premier temps définir des énumérations permettant d'identifier clairement : 1) Le niveau de l'erreur (`NEURON, LAYER, MODEL, LOSS`) dans un `enum` appelé `ErrorLevel` ainsi 2) qu'au niveau de la méthode dans laquelle l'erreur s'est produite (`FORWARD, BACKWARD, LOSS`) dans un `enum` appelé `ErrorMethode`.

2) Définissez ensuite la classe abstraite `AiException` dérivant de la classe `Exception` contenant les attributs `errorLevel` (de type `ErrorLevel`) et `errorMethode` (de type `ErrorMethode`). Ecrivez le constructeur à trois argument de `AiException` (`String message` pour la super classe `Exception` ainsi que pour initialiser les deux attributs de la classe). Enfin, redéfinir la méthode `getMessage` de la classe `Exception` en ajoutant au message de la méthode `getMessage` sans argument de la classe mère `Exception` les deux attributs de la classe `errorLevel` et `errorMethode` dans la chaîne de caractères retournée.

3) Ajoutez la classe `AiExceptionBackward` dérivant de la classe `AiException` contenant uniquement un constructeur contenant les arguments : message de type `String` pour le `message` (à attribuer à la classe `Exception` via le constructeur de `AiException`) et un argument de type `ErrorLevel` permettant de connaître à quel niveau l'erreur a été rencontrée dans une méthode de passe arrière (backward). Ces arguments seront utilisés pour initialiser les attributs de `AiException` via le constructeur de `AiException`.
Faites de même pour définir les classes `AiExceptionForward` et `AiExceptionLoss` en changeant uniquement dans les constructeurs respectifs, l'appel au super constructeur de la classe `AiException` en remplaçant la valeur de l'attribut `ErrorMethode` par `FORWARD` et `LOSS` respectivement. Lever des exceptions dans toutes les méthodes le nécessitant (`forward`, `backward` et `loss`) et modifier vos classes pour permettre une gestion des ces erreurs au niveau du `main` de la classe de `Test`.

D – Epilogue et tests …

4) Ecrire enfin la classe `Test` qui contiendra la méthode `main`. Cette méthode créera un modèle `MLP` et le testera sur différentes operations (générées aléatoirement dans une boucle). Commencez par une simple addition, soustraction puis multiplication. Dans ces trois cas, x ne contiendra que deux valeurs car l'opérande est fixé. Pour finir, testez avec la combinaison des trois opérandes avec x de taille 3 cette fois-ci car x contiendra également l'opérande.

IV) Dépôt et attendus du projet à rendre pour le **mardi 1 juin 2021**

Vous veillerez à associer un fichier par classe et suivre les bonnes pratiques de programmation vues en cours et TP. Par exemple, une variable, package ou méthode commence par une minuscule (maMethode, leCompteur, etc…), une variable statique en majuscule séparée par des _ (MON_COMPTEUR), le nom d'une classe commence par une majuscule (MaClasse). N'hésitez pas à commenter votre code (sans excès) et le rendre le plus intelligible possible avec des packages dédiés (voir figure ci-dessous).

Ce projet est **INDIVIDUEL**. Si vous avez nécessité l'aide d'un tiers, indiquez le clairement dans le rapport. Ce rapport, au **format PDF**, accompagnant vos **sources** dans **l'archive au format ZIP déposée sur la plateforme E-UAPV** avec vos nom et prénom (prenomNom.zip) contiendra les modélisations du projet et autres informations pertinentes (Introduction, Contexte, Difficultés rencontrées, Solutions apportées, Tests réalisés, Ajout de fonctionnalités, Conclusion, etc.).