Construire différentes topologies de réseaux de neurones en utilisant les layers prédéfinis de Tensorflow Keras

Eloise Molé Kamga et Hélène Toussaint, mars 2021

But : Le but de ce tutoriel est d'apprendre à construire différentes topologies de réseaux de neurones à l'aide des *layers* prédéfinis de **Tensorflow Keras**. Nous n'abordons pas la partie apprentissage, qui se traite de la même manière quelle que soit la topologie adoptée.

Prérequis : savoir construire un réseau de neurones basique à l'aide des API *Sequential* et *Functional* de **Tensorflow Keras**. Nous recommandons pour cela la lecture de [1] (introduction à **Tensorflow Keras** et à l'API *Sequential*) et [2] (utilisation de l'API *Functional* de **Tensorflow Keras**).

Remarques générales :

- dans la suite de ce document nous noterons **Tensorflow Keras** de manière abrégée : **tf.keras**,
- tous les exemples donnés dans ce document utilisent l'API *Functional* qui offre plus de souplesse que l'API *Sequential*,
- les scripts **Python** / **tf.keras** utilisés pour illustrer ce document sont tous à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip.

Table des matières

1 Rappels et vocabulaire

On suppose que les réseaux de neurones sont connus du lecteur, on rappelle néanmoins dans cette section quelques notions de base sur lesquelles s'appuie la suite du document.

1.1 Description générale et représentation graphique d'un réseau de neurones

Un réseau de neurones est généralement représenté comme une succession de « couches ». Chaque couche contient 1 ou plusieurs neurones. Les neurones d'une couche sont reliés aux neurones de la couche suivante de manière complète (**Figure 1**) ou partielle (**Figure 2**).

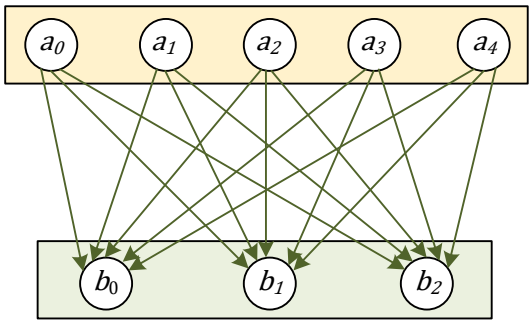

Figure 1. *Représentation d'une couche de 5 neurones <u>complétement</u> reliée à une couche de 3 neurones*

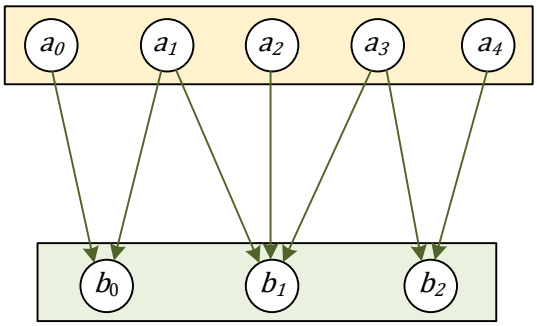

Figure 2. *Représentation d'une couche de 5 neurones <u>partiellement</u> reliée à une couche de 3 neurones*

Chaque neurone j a une valeur d'entrée $in(j)$ (aussi appelée *valeur de préactivation*) et une valeur de sortie $out(j)$ (aussi appelée *valeur d'activation*). La valeur d'entrée d'un neurone est égale à la somme des valeurs de sortie des neurones le précédant, pondérée par des poids, à laquelle on ajoute un *biais.* Les poids sont souvent représentés sur les arcs et le biais est souvent représenté comme un neurone supplémentaire, de valeur de sortie 1 (fixe) avec un poids B (**Figure 3**). La valeur de sortie d'un neurone est égale à sa valeur d'entrée à laquelle on applique une fonction dite *d'activation*. La **Figure 4** donne les valeurs d'entrée et de sortie du neurone b de la **Figure 3** avec une fonction d'activation ϕ.

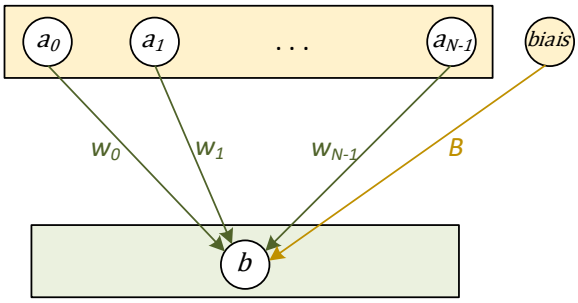

Figure 3. *Représentation des poids et du biais*

$$in(b) = B + \sum_{i=0}^{N-1} out(a_i).w_i$$

$$out(b) = \phi\big(in(b)\big) = \phi\left(B + \sum_{i=0}^{N-1} out(a_i).w_i\right)$$

Figure 4. *Calcul des valeurs d'entrée et de sortie du neurone b du réseau de la **Figure 3***

Dans le cas général, il y a plusieurs neurones par couche et les poids utilisés pour calculer la valeur d'entrée d'un neurone sont différents d'un neurone à l'autre, ainsi que les biais (voir **Figure 5**). On a pour habitude de regrouper ces poids dans une matrice (**Figure 6**).

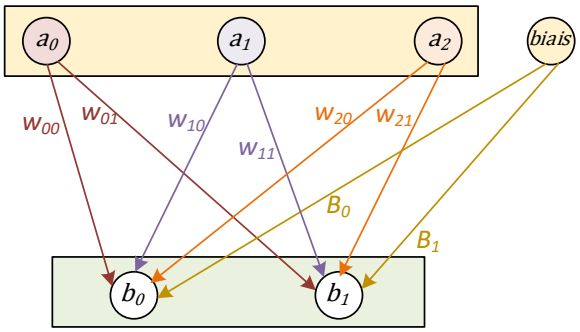

Figure 5. *Poids et biais entre deux couches de neurones non unitaires*

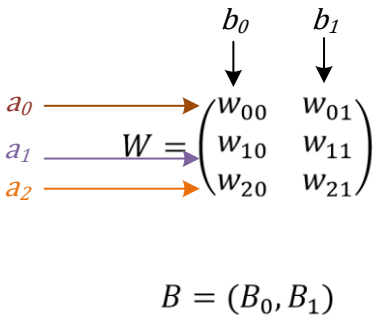

Figure 6. *Matrice des poids W et vecteur des biais B associés à la* ***Figure 5***

1.2 Vocabulaire

Dans la suite nous utiliserons le vocabulaire suivant :

- le terme *couche* sera utilisé pour la description générale d'un réseau alors que le terme *layer* sera utilisé pour faire référence à une couche particulière prédéfinie de **tf.keras** (voir [3]),
- les termes *modèle* et *réseau* seront utilisés indistinctement (*modèle* étant le vocabulaire utilisé dans **tf.keras** pour faire référence à un réseau de neurones).

2 Les couches et les poids dans tf.keras

Dans **tf.keras** il existe différentes couches, appelées *layers*, que l'on peut utiliser facilement pour créer un réseau de neurones (également appelé *modèle*). Ces *layers* définissent la manière dont les neurones reçoivent l'information des *layers* précédents.

2.1 Le *layer Dense*

Les exemples de code de cette section sont extraits du script tutoPoids.py qui peut être téléchargé à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip

Le *layer Dense* (voir [5]) est le *layer* de référence pour créer des couches dont les neurones reçoivent en entrée une valeur égale à la somme pondérée de l'ensemble des neurones du *layer* précédent

(comme dans la **Figure 5**). Le *layer Dense* permet donc de créer une couche qui est reliée de manière complète à la couche précédente.

Le code suivant montre comment créer le réseau de la **Figure 5**.

```
#création de la couche d'input (avec 3 neurones, car les entrées sont de taille 3)
my_input = keras.Input(shape=(3,), name="input")

# création de la couche de sortie avec 2 neurones
my_output = keras.layers.Dense(2, activation='linear', name="output")(my_input)

#construction du modèle
model = keras.Model(inputs=my_input, outputs=my_output)
```

Figure 7. Extrait du code pour construire le réseau de la **Figure 5**

Dans **tf.keras**, les couches sont représentées par une matrice des poids et un vecteur des biais.

La fonction suivante permet d'afficher les poids d'une couche. Elle prend en paramètre le modèle et le nom de la couche dont on souhaite afficher les poids.

```
def affichePoids(model, name):

        print ("affichage des poids de la couche", name)

        #afficher les poids qui « arrivent » sur une couche
        WS = model.get_layer(name).get_weights()

        print("weights = ", WS[0] )
        print("biais = ", WS[1] )
```

Par exemple, si on souhaite afficher les poids de la couche de sortie du modèle de la **Figure 7** :

```
affichePoids(model, "my_output")
```

Les poids sont alors affichés de la manière suivante :

```
affichage des poids de la couche my_output

weights =  [[1.        4.]
            [2.        5.]
            [3.        6.]]
biais =  [10.      20.]
```

Figure 8. *Affichage des poids*

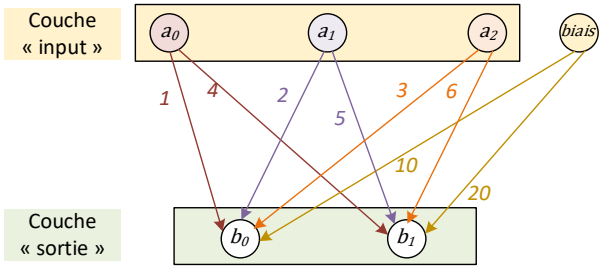

Figure 9. *Représentation graphique des poids associés à la* ***Figure 8***

Ils signifient que les valeurs d'entrée (ou de préactivation) pour les 2 neurones (b_0, b_1) de la couche de sortie sont calculées de la manière suivante (pour simplifier les notations $out(a_i)$ est remplacé par $a_i, i = \{0,1,2\}$) :

$in(b_0) = a_0 + 2a_1 + 3a_2 + 10$

$in(b_1) = 4a_0 + 5a_1 + 6a_2 + 20$

2.2 Initialisation des poids du *layer Dense*

Les exemples de code de cette section sont extraits du script tutoPoids.py qui peut être téléchargé à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip

L'initialisation des poids est un élément important qui joue un rôle clé dans la descente du gradient. Par défaut cette initialisation est réalisée par **tf.keras** de manière aléatoire. Cependant, si on a une idée des poids finaux, il peut être intéressant de réaliser cette initialisation nous-même. Il suffit pour cela de passer en paramètre la matrice des poids et le vecteur des biais lors de la création du *layer Dense*.

Le code suivant montre comment construire le réseau de la **Figure 9** en initialisant les poids (y compris les biais) à la main.

```
# création de la couche d'input (avec 3 neurones)
my_input = keras.Input(shape=(3,), name="input")

# création de la couche de sortie avec un 2 neurones et initialisation des poids
my_biais = np.array([10, 20])
my_poids = np.array([[1, 4], [2, 5], [3, 6]])
my_output = keras.layers.Dense(2, activation='linear', weights=[my_poids, my_biais],
name="output")(my_input)

# construction du modèle
model = keras.Model(inputs=my_input, outputs=my_output)

# dessin du modele
keras.utils.plot_model(model, "Func_tutoPoids2.png", show_shapes=True, dpi=192)
```

2.3 Différentes manières de fusionner les couches : *Merging layers*

Les exemples de code de cette section sont extraits du script tutoMerge.py qui peut être téléchargé à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip

tf.keras offre la possibilité de fusionner, de différentes manières, une ou plusieurs couches d'un réseau. Pour cela il faut utiliser des *layers* particuliers appelés *merging layers* [6].

2.3.1 Concaténation de plusieurs couches

La manière la plus simple de fusionner des couches est de les concaténer : dans ce cas les neurones sont simplement recopiés de leur couche initiale vers la couche de concaténation sans modification de leur valeur de sortie.

Le code suivant montre comment concaténer 3 couches à l'aide du *layer Concatenate* [7].

```
# création de 3 couches d'entrée de taille respective 3, 2 et 4
my_input1 = keras.Input(shape=(3,), name="input1")
```

```
my_input2 = keras.Input(shape=(2,), name="input2")
my_input3 = keras.Input(shape=(4,), name="input3")

# concaténation des 3 couches d'entrée
concat = keras.layers.concatenate([my_input1, my_input2, my_input3])

# création de la couche de sortie avec un 2 neurones
my_output = keras.layers.Dense(2, activation='linear', name="output")(concat)

# construction du modèle
model = keras.Model(inputs=[my_input1, my_input2, my_input3], outputs=my_output)
```

Figure 10. *Exemple d'utilisation du layer concatenate*

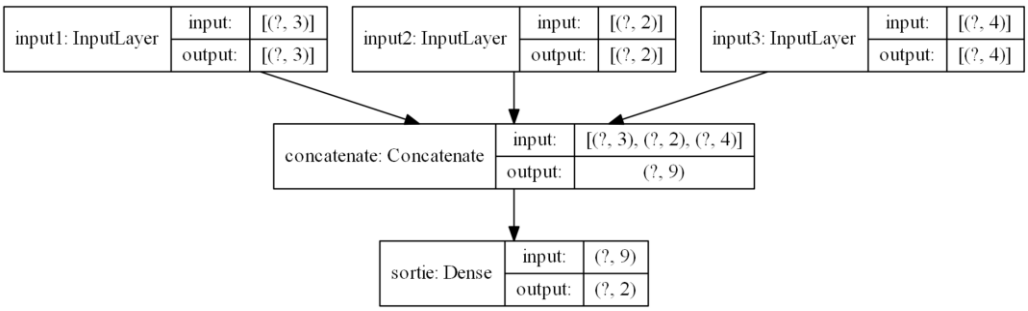

Figure 11. *Réseau créé avec le code de la* ***Figure 10***

2.3.2 Multiplication terme à terme de deux couches

Un autre *layer* très intéressant est le *layer Multiply* [8]. Il permet de multiplier la valeur de sortie de 2 couches de neurones terme à terme. L'extrait de code suivant illustre son utilisation :

```
# création de 2 couches d'entrée de taille 3
my_input1 = keras.Input(shape=(3,), name="input1")
my_input2 = keras.Input(shape=(3,), name="input2")

# une couche cachée par couche d'entrée
couche1 = keras.layers.Dense(3, activation='relu', name="C1")(my_input1)
couche2 = keras.layers.Dense(3, activation='relu', name="C2")(my_input2)

# multiplication des 2 couches précédentes
my_output = keras.layers.Multiply()([couche1, couche2])
```

Figure 12. *Exemple d'utilisation du layer Multiply*

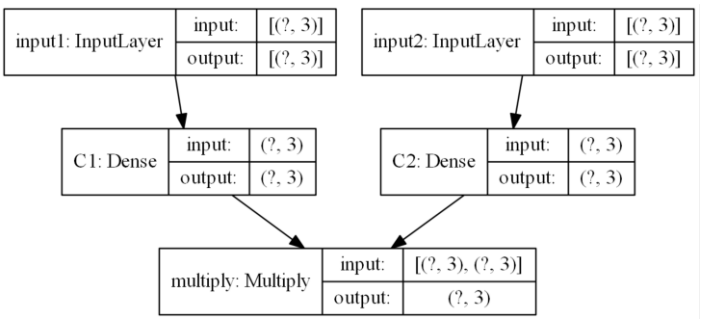

Figure 13. *Réseau créé avec le code de la* ***Figure 12***

2.3.3 Afficher la sortie des couches internes d'un modèle

Afin de s'assurer que les couches ont été fusionnées comme on l'entend, nous avons besoin d'accéder à la sortie des couches internes d'un modèle. Pour cela il faut :

1. Récupérer la couche du modèle concernée à l'aide de son nom (il faut donc lui avoir donné un nom lors de sa création),
2. Appeler la couche avec les *inputs* du modèle pour lesquels on souhaite connaître la sortie.

Le code suivant illustre ceci sur le modèle créé **Figure 12**.

```
# exemple de données pour les 2 couches input
in1_test = np.asarray([[1,2,3]])
in2_test = np.asarray([[4,5,6]])

# on recupère les couches internes à partir de leur nom
c1_layer = keras.Model(inputs=model.input, outputs=model.get_layer('C1').output)
c2_layer = keras.Model(inputs=model.input, outputs=model.get_layer('C2').output)

# on donne aux couches internes les inputs souhaités
couche1_out = c1_layer({'input1' : in1_test, 'input2':  in2_test})
couche2_out = c2_layer({'input1' : in1_test, 'input2':  in2_test})

# on recupère la sortie du modèle avec les inputs souhaités
output_out = model.predict({'input1' : in1_test, 'input2':  in2_test})

# on affiche la sortie des différentes couches
print('couche1 : ', couche1_out)
print('couche2 : ', couche2_out)
print('output : ', output_out)
```

Figure 14. *Extrait de code permettant d'afficher la sortie de différentes couches*

On obtient, par exemple, l'affichage suivant :

```
couche1 : tf.Tensor([[-1.399934  -2.0355947  3.6685464]], shape=(1, 3), dtype=float32)
couche2 : tf.Tensor([[0.         4.4896154  0.15367174]], shape=(1, 3), dtype=float32)
output : [[-0.         -9.139037    0.56375194]]
```

On constate que la couche de sortie est bien la multiplication terme à terme des couches internes 1 et 2.

NB. Si vous testez ce code vous n'obtiendrez probablement pas la même chose, les poids étant initialisés de manière aléatoire.

3 Relier deux couches de neurones de manière partielle

Les exemples de code de cette section sont extraits du script tutoArcPartiel.py qui peut être téléchargé à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip

3.1 Problématique générale

Il n'existe pas, à notre connaissance, de layer prédéfini qui permette de relier 2 couches de manière partielle (c'est-à-dire telle que présentée dans la ***Figure 2***).

Une solution possible serait de créer son propre *layer* (ce qui est réalisable dans **tf.keras** en créant une classe dérivée de la classe *Layer*, voir [9]). Cependant cette technique demande une très bonne compréhension des différents éléments qui forment un *layer* au sens de **tf.keras** ainsi que de la manière dont ils s'articulent. Elle semble, par conséquent, assez fastidieuse à mettre en place.

Une autre solution, qui est celle que nous avons choisis de présenter ici, consiste à utiliser les *layers* de **tf.keras** prédéfinis et à les combiner de manière à obtenir exactement la topologie de réseau souhaitée. C'est l'objet de la prochaine sous-section.

3.2 Combiner des *layers* prédéfinis de **tf.keras** pour relier 2 couches de manière partielle

Remarque : dans cette section on ne s'intéresse qu'aux arcs (et donc qu'aux poids) entre 2 couches de neurones. Le biais n'est pas pris en compte, et, même s'il existe, on ne le représente pas sur les schémas afin de ne pas surcharger les illustrations.

3.2.1 Principe général

L'idée générale est la suivante : supposons que l'on souhaite créer une couche A de N neurones $(a_0, \dots, a_{N-1})$ qui envoie de l'information à une couche B de M neurones $(b_0, \dots, b_{M-1})$. De plus nous souhaitons choisir exactement quel neurone de la couche A envoie de l'information à quel neurone de la couche B (**Figure 15** (a)). Nous avons donc besoin de pouvoir manipuler chaque neurone de manière indépendante. Nous réalisons cela en 4 étapes (voir (**Figure 15** (b)) :

Etape 1. Créer N « sous-couches » unitaires $A_0, \dots, A_{N-1}$ au lieu d'une couche de N neurones : la sous-couche $A_{i,(i=0..N-1)}$ contient uniquement le neurone a_i.

Etape 2. Pour chaque neurone $b_{j,(j=0..M-1)}$ de la couche B, identifier les neurones $a_{i_1}, \dots, a_{i_K} (0 \leq i_1 \leq i_K \leq N-1)$ qui envoient de l'information à b_j. Concaténer les sous-couches $A_{i_1}, \dots, A_{i_K}$ correspondantes. On obtient ainsi M nouvelles sous-couches $(C_0, \dots, C_{M-1})$ de type *Concatenate.* Notons qu'un même neurone $a_{i\,(i=0..N-1)}$ peut se retrouver dans plusieurs sous-couches *Concatenate* (sa valeur sera la même dans toutes les sous-couches).

Etape 3. Créer M sous-couches unitaires $(B_0, \dots, B_{M-1})$ telles qu'une sous-couche $B_{j\,(j=0..M-1)}$ contient uniquement le neurone b_j et reçoit en entrée la sortie de la sous-couche C_j.

Etape 4. Pour finir, concaténer les sous-couches unitaires $(B_0, \dots, B_{M-1})$ de sorte de retrouver une couche de M neurones $(b_0, \dots, b_{M-1})$.

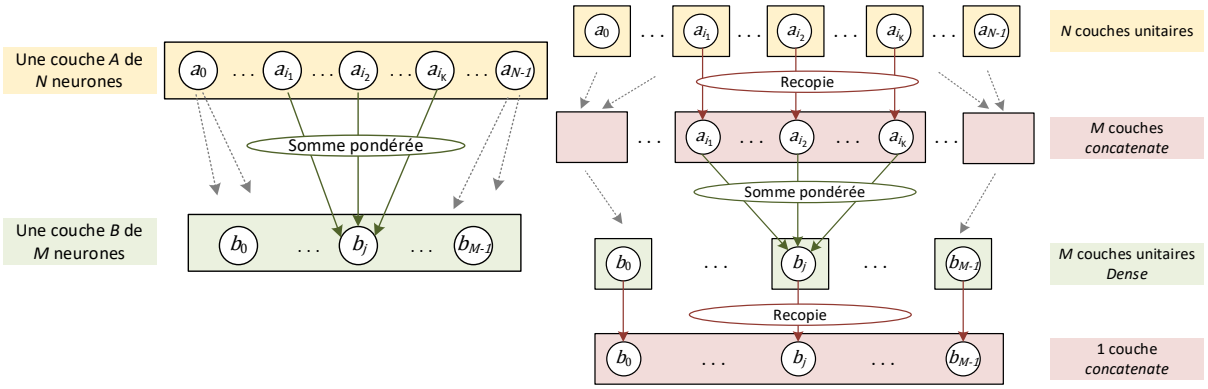

Figure 15. *(a) Représentation d'un réseau avec 2 couches partiellement reliées – (b) Représentation du même réseau en utilisant exclusivement des layers* ***tf.keras*** *Dense et Concatenate*

Remarque : Les entiers N et M peuvent être potentiellement grands et/ou inconnus au moment où on écrit le code (par exemple si on les lit dans un fichier). C'est pourquoi il semble judicieux d'utiliser des tableaux pour stocker les différentes *sous-couches*.

La sous-section suivante illustre ceci sur un exemple.

3.2.2 Illustration sur un exemple

On souhaite représenter le réseau de la ***Figure 2*** (rappelé ci-dessous : **Figure 16** (a)) en utilisant uniquement des *layers Dense* et *Concatenate* et en suivant la méthodologie donnée en section 3.2.1.

Le résultat obtenu est représenté **Figure 16** (b).

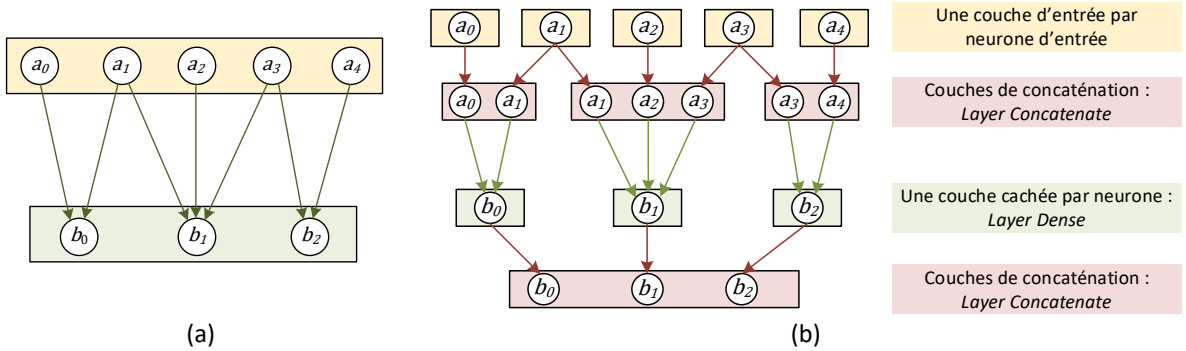

Figure 16. *(a) réseau initial – (b) le même réseau représenté avec des layers Dense et Concatenate*

Le code qui permet de programmer ce réseau est donné ci-dessous.

```
# ETAPE 2.1 : création des sous-couches A_0 ... A_{N-1} (ici N = 5)

N = 5
ssCoucheA = []

for i in range(N):
        tmp = keras.Input(shape=(1,), name="input"+str(i))
        ssCoucheA.append(tmp)
```

```
# ETAPE 2.2 : on crée les M = 3 couches de concaténation

ssCoucheC = []

ssCoucheC.append(keras.layers.concatenate([ssCoucheA[0], ssCoucheA[1]]))
ssCoucheC.append(keras.layers.concatenate([ssCoucheA[1], ssCoucheA[2], ssCoucheA[3]]))
ssCoucheC.append(keras.layers.concatenate([ssCoucheA[3], ssCoucheA[4]]))


# ETAPE 2.3 :  création des sous-couches B_0 ... B_{M-1} (ici M = 3)
ssCoucheB = []
M = len(ssCoucheC)
for i in range(M):
       ssCoucheB.append(keras.layers.Dense(1, activation='linear')(ssCoucheC[i]))

# ETAPE 2.4 :  on concatene les sous-couches B
my_output = keras.layers.concatenate(ssCoucheB)

# construction du modèle : définition des couches d'entrée et de sortie
model = keras.Model(inputs=ssCoucheA, outputs=my_output)

# dessin du modele
keras.utils.plot_model(model, "reseauFunctionalSparse.png", show_shapes=True, dpi=192)
```

Figure 17. *Extrait du code qui permet de construire le modèle de la* ***Figure 16***

Le schéma suivant montre le résultat de la fonction **`plot_model`** du code précédent : c'est la représentation schématique par **tf.keras** du modèle de la **Figure 16** (b).

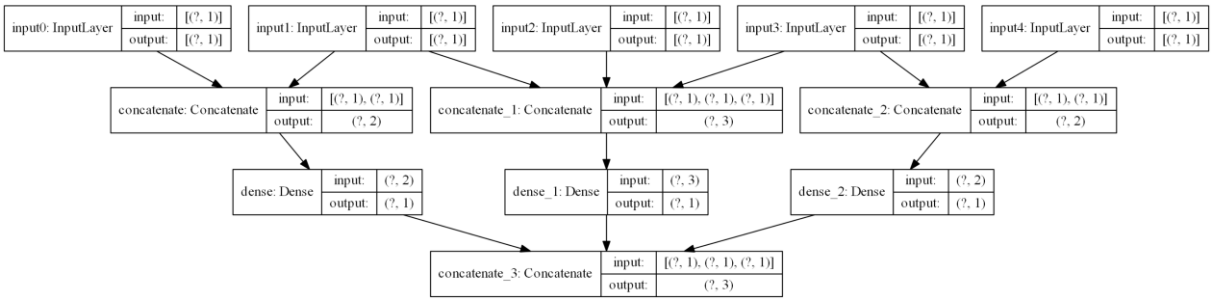

Figure 18. *Représentation synthétique du modèle de la* ***Figure 16*** *(b) (obtenue par la fonction* `plot_model` *[6])*

4 Fixer des poids dépendant des instances

Les exemples de code de cette section sont extraits du script tutoVal.py qui peut être téléchargé à l'adresse https://perso.limos.fr/~hetoussa/doc/codeTutoKeras.zip

Le but de cette section est de montrer comment on peut prendre en compte, à l'intérieur d'un réseau, et de manière explicite, des valeurs propres à chaque instance.

Pour fixer les idées, nous allons illustrer notre propos sur le réseau de la

Figure *19*. Dans cet exemple nous supposons qu'une entrée est formée de deux types de valeurs :

- N valeurs $a_0, a_1, \dots, a_{N-1}$

- M coefficients $\lambda_0, \lambda_1, \dots, \lambda_{M-1}$

Les valeurs $a_0, a_1, \dots, a_{N-1}$ sont les entrées du réseau alors que les coefficients $\lambda_0, \lambda_1, \dots, \lambda_{M-1}$ sont utilisés uniquement pour calculer la valeur de sortie du réseau s qui est telle que :

$$s = \sum_{j=0}^{M-1} \lambda_j . out(b_j)$$

où $b_{j,(j=0,\dots,M-1)}$ est le $j^{ème}$ neurone de la couche précédant la sortie du réseau (voir

Figure *19*).

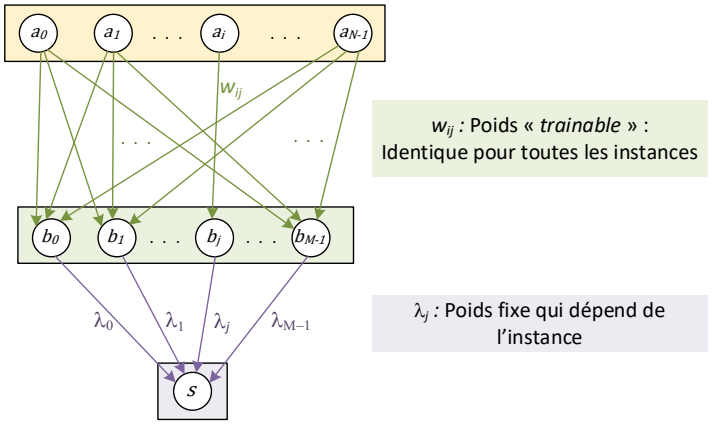

***Figure 19**. Exemple de réseau dans lequel des poids fixes et dépendant de l'instance interviennent*

Nous avons vu dans la section 2 comment fixer les poids d'un *layer Dense*. Malheureusement, cette technique ne fonctionne que pour les poids « classiques » : c'est-à-dire qui sont les mêmes pour toutes les instances.

Pour personnaliser les poids en fonction de chaque instance il faut les donner au réseau comme une entrée (c'est-à-dire dans un *layer Input*) puis utiliser des *layers* spéciaux (par exemple le *layer Multiply*, voir section 2) qui permettent de combiner différentes couches entre-elles.

Pour programmer le réseau de la

Figure *19*, il faut donc le transformer en utilisant par exemple des *layers Dense* et *Multiply*, comme illustré dans la **Figure 20**.

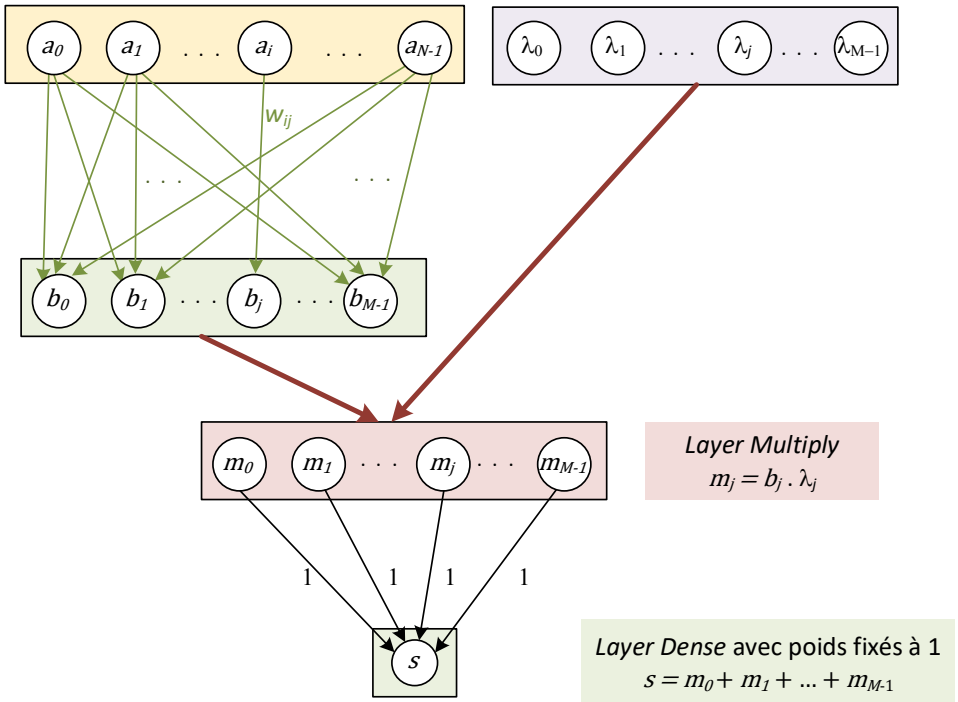

Figure 20. *Réseau de la*

***Figure* 19** *reformulé avec des layers Dense et Multiply*

L'extrait de code suivant montre comment programmer le réseau de la **Figure 20**.

```
N = 8
M = 5

# création de 2 couches d'entrée de taille respective N et M
my_input1 = keras.Input(shape=(N,), name='input1')
my_input2 = keras.Input(shape=(M,), name='input2')

# une couche cachée
couche1 = keras.layers.Dense(M, activation='linear', name='C1')(my_input1)

# multiplication de la couche d'entrée 'input2' et de la couche cachée
mult = keras.layers.Multiply()([my_input2, couche1])

# 1 neurone de sortie = somme des sorties de la couche précédente
# on a donc des fixes poids = 1 et un biais = 0
my_biais = np.asarray([0])
my_poids = np.ones((M,1))
my_output = keras.layers.Dense(1, activation='linear', name='out', weights=[my_poids,
my_biais], trainable=False)(mult)

# création du modèle
model = keras.Model(inputs=[my_input1, my_input2], outputs=my_output)
```

Figure 21. *Code pour créer le modèle de la* **Figure 20**

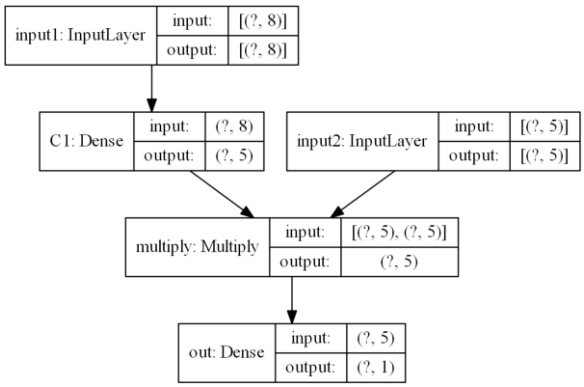

Figure 22**. Schéma du modèle créé par le code de la **Figure 21

5 Références

[1] Jason Brownlee. **TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras**. https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/

[2] Jason Brownlee. **How to Use the Keras Functional API for Deep Learning**. https://machinelearningmastery.com/keras-functional-api-deep-learning/

[3] Keras API reference. **Layer**. https://keras.io/api/layers/

[4] Keras API reference. **Model plotting utilities.** https://keras.io/api/utils/model_plotting_utils/

[5] Keras API reference. **Dense layer.** https://keras.io/api/layers/core_layers/dense/

[6] Keras API reference. **Merging layer.** https://keras.io/api/layers/merging_layers/

[7] Keras API reference. **Concatenate layer.** https://keras.io/api/layers/merging_layers/concatenate/

[8] Keras API reference. **Multiply layer.** https://keras.io/api/layers/merging_layers/multiply/

[9] Keras API reference. **Making new layers and models via subclassing.** https://keras.io/guides/making_new_layers_and_models_via_subclassing/