UNIVERSITÉ NATIONALE DU VIETNAM À HANOÏ

INSTITUT FRANCOPHONE INTERNATIONAL

Option : Systèmes Intelligents et Multimédia (SIM)

Promotion : XXII

APPRENTISSAGE AUTOMATIQUE

TP2 : APPRENTISSAGE PROFOND

Jean Claude SERUTI ZAGABE
Azaria ALLY SAIDI
Hugues MADIMBA KANDA
MASTER II

Encadrant :

Dr. Thanh-Nghi Do

Année académique 2018-2019

Table des matières

0 Introduction

0.1 Contexte

Vu la découverte des librairies de Deep Learning Tensorflow / Keras pour Python, l'implémentation de perceptrons simples et multicouches seront traités dans les différents problèmes de classement (apprentissage supervisé). Tensorflow est une bibliothèque open-source développée par l'équipe Google Brain qui l'utilisait initialement en interne. Elle implémente des méthodes d'apprentissage automatique basées sur le principe des réseaux de neurones profonds (deep learning). Nous pouvons l'exploiter directement dans un programme rédigé en Python grace à l'API.

Keras est une librairie Python qui encapsule l'accès aux fonctions proposées par plusieurs librairies de machine learning, en particulier Tensorflow. De fait, Keras n'implémente pas nativement les méthodes. Elle sert d'interface avec Tensorflow simplement, parce qu'elle nous facilite grandement la vie en proposant des fonctions et procédures relativement simples à mettre en œuvre.

0.2 Objectif

Objectif de ce TP2 est la prise en main des outils. Pour aller à l'essentiel, nous implémenterons des perceptrons simples et multicouches en python selon les problèmes posés.

1 Perceptron simple, deux classes

Etant donnée :

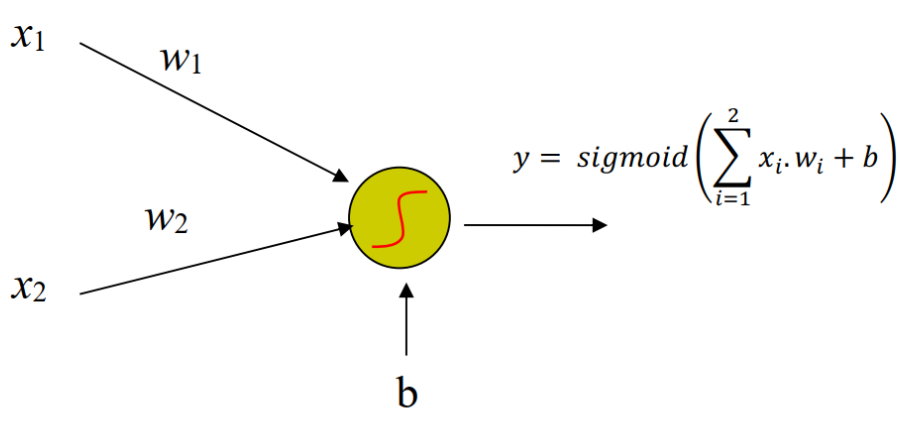

FIGURE *1 – Perceptron simple à deux classes*

1.1 Apprentissage d'un perceptron simple

Nous répétons les étapes fournies avec tensorflow dans l'énoncer afin de reportez les résultats suivants :

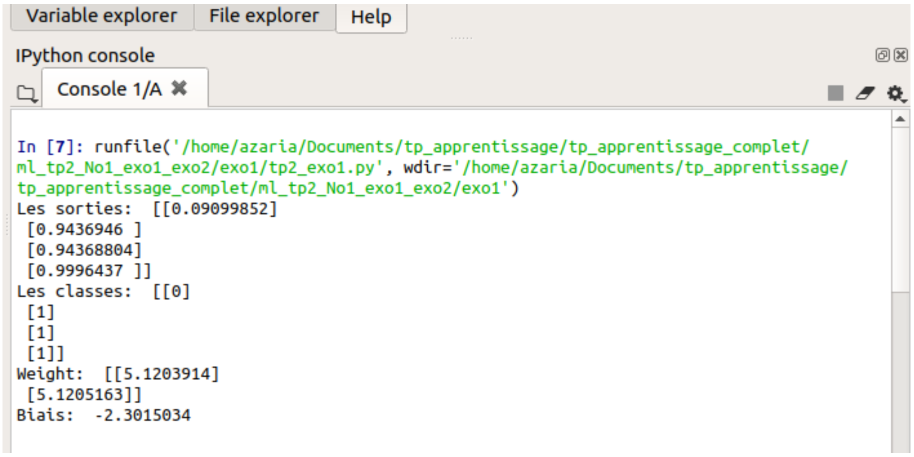

Figure 2 – *Résultats*

Et puis nous visualisons les données d'apprentissage et la ligne droite obtenue comme nous montre la figure 3.

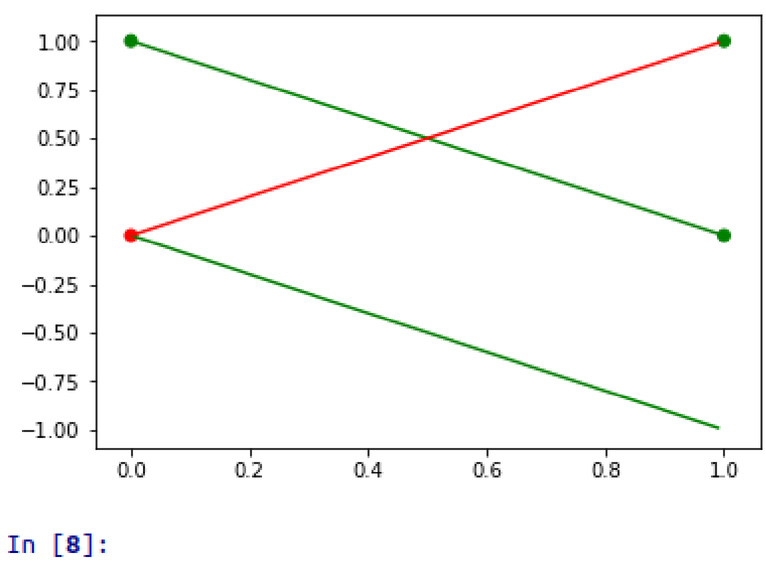

Figure 3 – *Visualisation graphique des données*

Cette visualisation est obtenue grâce à librairie matplolib de python.

Rappelons que l'équation de la ligne droite du perceptron est : $w_1.x_1 + w_2.x_2 + b = 0$

Donc, la droite trouvée est :

$$-2.3015034 + 5.1203914.x_1 + 5.1205163.x_2 = 0 \tag{1}$$

contenant les points : $P(0, 5.1205163)$ et $Q(5.1203914, 0)$

Au regard de résultat précèdent, les données sont linéairement séparables et qu'il y a trois supports de vecteurs à savoir : $S_1(0,0)$, $S_2(0,1)$ et $S_3(1,0)$.

Notons $\varphi()$ la fonction de mapping et d'identité S_i' le support de vecteur i augmenté de la valeur 1. Nous utiliserons les vecteurs augmentés afin de tenir compte du biais. L'apprentissage par l'algorithme SVM revient donc à déduire les coefficients α_1,α_2, α_3 qui solutionnent les systèmes suivants :

$$\begin{aligned}
\alpha_1\varphi(S_1)*\varphi(S_1) + \alpha_2\varphi(S_2)*\varphi(S_1) + \alpha_3\varphi(S_3)*\varphi(S_1) = 0\\
\alpha_1\varphi(S_1)*\varphi(S_2) + \alpha_2\varphi(S_2)*\varphi(S_2) + \alpha_3\varphi(S_3)*\varphi(S_2) = 1\\
\alpha_1\varphi(S_1)*\varphi(S_3) + \alpha_2\varphi(S_2)*\varphi(S_3) + \alpha_3\varphi(S_3)*\varphi(S_3) = 1
\end{aligned}$$

Puisque nous savons que $\varphi() = 1$ le système peut être réécrit sous la forme suivante :

$$\begin{aligned}
\alpha_1\text{S'}_1*S'_1 + \alpha_2\text{S'}_2*S'_1 + \alpha_3\text{S'}_3*S_1 = 0\\
\alpha_1\text{S'}_1*S_2 + \alpha_2\text{S'}_2*S'_2 + \alpha_3 S'_3*S'_2 = 1\\
\alpha_1 S'_1*S'_3 + \alpha_2\text{S'}_2*S'_3 + \alpha_3\text{S'}_3*S'_3 = 1
\end{aligned}$$

Avec $S'_1(0,0,1)$, $S'_2(0,1,1)$ et $S'_3(1,0,1)$. En remplaçant les S'_i par leur valeur nous obtenons :

$$\begin{aligned}
\alpha_1 + \alpha_2 + \alpha_3 = 0\\
\alpha_1 + 2\alpha_2 + \alpha_3 = 1\\
\alpha_1 + \alpha_2 + 2\alpha_3 = 1
\end{aligned}$$

Ce système à comme solution : α_1 = -5 ;α_2 = 2 ; α_3 = 2
On sait que w'= $\sum \alpha_i$S'$_i$ d'où

$$w' = -5(0,0,1) + 2(0,1,1) + 2(1,0,1) = (2,2,-1)$$

w' se décompose en w = (2,2) et b = -1 ce qui permet d'écrire l'équation de la droite optimale séparant nos données.

1.2 Implémentation d'algorithme de Perceptron en Python (utilisant Tensorflow)

En implémentant l'algorithme de perceptron en python, nous passons à l'utilisation de notre jeu de données leukemia avec comme résultat obtenu ci dessous : Cette figure '4 illustre une précision faible de ce modèle. La figure 5 illustré ici n'arrive pas à atteindre une précision souhaitée.

Nous remarquons également que pour ces deux bases, le programme apprend et classe très bien les données. Nous concluons donc que notre implémentation du perceptron simple est correcte.

```
Epoch 502  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 503  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 504  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 505  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 506  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 507  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 508  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 509  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 510  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 511  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 512  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 513  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 514  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 515  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 516  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 517  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 518  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 519  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 520  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 521  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 522  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 523  Loss:  0.7105263   Accuracy:  0.28947368
```

FIGURE *4 – Resultat Leukemia*

```
Epoch 982  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 983  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 984  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 985  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 986  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 987  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 988  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 989  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 990  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 991  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 992  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 993  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 994  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 995  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 996  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 997  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 998  Loss:  0.7105263   Accuracy:  0.28947368
Epoch 999  Loss:  0.7105263   Accuracy:  0.28947368

In [10]:
```

FIGURE *5 – Resultat spam*

2 Perceptron simple, multi-classes

Quand le nombre de classes est supérieure à 2, il faut utiliser C neurones sortis (C : nombre de de classes),

2.1 Construction d'un réseau de neurones simple, multi-classes

Nous construisons le graphe en se basant de l'énoncer et de l'exemple issu du TP, en implémentant la fonction de perte qui inclus les trois classes par l'encodage one-hot, avec les étiquettes de nos données de la base iris qui consiste à classer (Iris-setosa, Iris-versicolor, Iris-virginica), tout en utilisant AdamOptimizer.

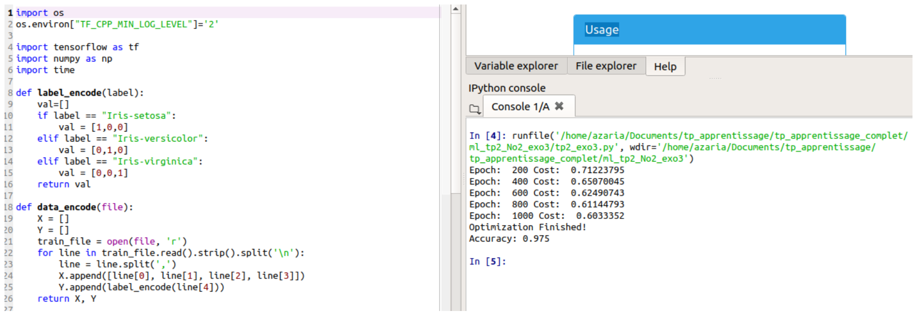

FIGURE 6 – *Résultat d'iris*

On remarque qu'on a une forte précision de notre modèle qui utilise le framework Tensorflow.

3 Perceptron multicouche

Quand des données ne sont pas linéairement séparables, on utilise plusieurs couches au lien d'une seule, comme exemple de la table 1

On essayera donc de rajouter une couche intermédiaire qui s'appelle la couche cachée, comme illustré sur cette figure 7

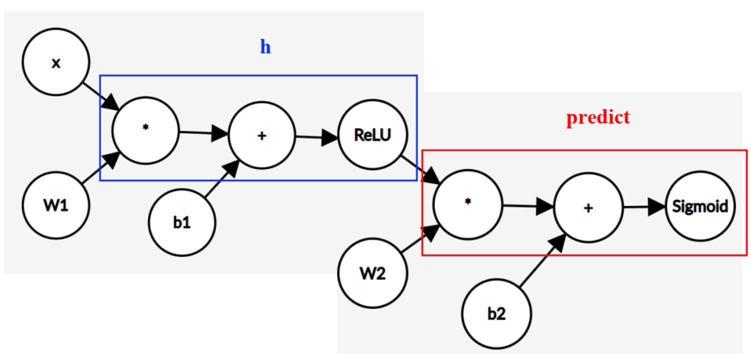

FIGURE 7 – *Couche cachée*

X1	X2	Classe
0.204000	0.834000	0
0.222000	0.730000	0
0.298000	0.822000	0
0.450000	0.842000	0
0.412000	0.732000	0
0.298000	0.640000	0
0.588000	0.298000	0
0.554000	0.398000	0
0.670000	0.466000	0
0.834000	0.426000	0
0.724000	0.368000	0
0.790000	0.262000	0
0.824000	0.338000	0
0.136000	0.260000	1
0.146000	0.374000	1
0.258000	0.422000	1
0.292000	0.282000	1
0.478000	0.568000	1
0.654000	0.776000	1
0.786000	0.758000	1
0.690000	0.628000	1
0.736000	0.786000	1
0.574000	0.742000	1

TABLE *1 – Ensemble de deux Classe*

3.1 Compléter ce programme en rajoutant : fonction de perte, entrainement

Dans cet exercice on implémente la fonction de perte et on entraine le réseau puis on calcul la précision (accuracy) du modèle d'apprentissage, on va essayer en ajoutant le nombre d'itération à 1000 et le pas d'apprentissage 0.1.

FIGURE *8 – itération de la Base*

Après cette figure 8 on applique la fonction de perte pour obtenir les classes prédites ci dessous, la figure 9 :

```
 [0.43478256]
 [0.43478256]
 [0.43478256]]
Prediction:
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
```

Figure 9 – *Classes prédites*

Nous pourrions bien visualiser les résultats obtenues de la figure 9 après chaque itération d'une manière graphique.

```
 [0.]]
```

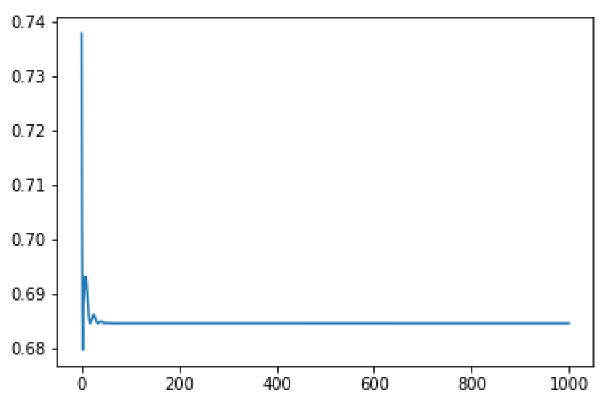

Figure 10 – *Visualisation graphique*

4 Apprentissage profond avec Keras

4.1 Perceptron simple, deux classes

Dans cette phase de tache, nous allons exploiter la bibliothèque Keras afin d'exploiter la tache qui nous ai demandé dans l'énoncer suivant les codes exemplaires.

```
In [15]: runfile('/home/azaria/Documents/tp_apprentissage/tp_apprentissage_complet/ml_tp2_No4_exo5/
exemple_4a_4b/tp2_4_a_MLP_Keras.py', wdir='/home/azaria/Documents/tp_apprentissage/tp_apprentissage_complet/
ml_tp2_No4_exo5/exemple_4a_4b')
Using TensorFlow backend.
Epoch 1/5
4/4 [==============================] - 1s 182ms/sample - loss: 1.0503 - acc: 0.2500
Epoch 2/5
4/4 [==============================] - 0s 1ms/sample - loss: 1.0490 - acc: 0.5000
Epoch 3/5
4/4 [==============================] - 0s 1ms/sample - loss: 1.0477 - acc: 0.5000
Epoch 4/5
4/4 [==============================] - 0s 1ms/sample - loss: 1.0464 - acc: 0.5000
Epoch 5/5
4/4 [==============================] - 0s 1ms/sample - loss: 1.0451 - acc: 0.5000
[array([[-0.02854896],
       [-0.51897335]], dtype=float32), array([0.], dtype=float32)]

In [16]:
```

FIGURE *11 – Construction de modèle*

Passons maintenant au changement comme le montre le résultat de la figure 12

```
In [17]: runfile('/home/azaria/Documents/tp_apprentissage/tp_apprentissage_complet/ml_tp2_No4_exo5/
exemple_4a_4b/tp2_4_b_MLP_Keras.py', wdir='/home/azaria/Documents/tp_apprentissage/tp_apprentissage_complet/
ml_tp2_No4_exo5/exemple_4a_4b')
Epoch 1/100
8/8 [==============================] - 0s 16ms/sample - loss: 1.3583 - acc: 0.3750
Epoch 2/100
8/8 [==============================] - 0s 380us/sample - loss: 1.3562 - acc: 0.1250
Epoch 3/100
8/8 [==============================] - 0s 204us/sample - loss: 1.3541 - acc: 0.1250
Epoch 4/100
8/8 [==============================] - 0s 186us/sample - loss: 1.3520 - acc: 0.1250
Epoch 5/100
8/8 [==============================] - 0s 372us/sample - loss: 1.3499 - acc: 0.1250
Epoch 6/100
8/8 [==============================] - 0s 235us/sample - loss: 1.3478 - acc: 0.1250
Epoch 7/100
8/8 [==============================] - 0s 179us/sample - loss: 1.3457 - acc: 0.1250
Epoch 8/100
8/8 [==============================] - 0s 449us/sample - loss: 1.3437 - acc: 0.1250
Epoch 9/100
8/8 [==============================] - 0s 163us/sample - loss: 1.3416 - acc: 0.1250
```

FIGURE *12 – Construction de modèle100*

Une suite pour le poids récupérer les poids dans cette 13

```
8/8 [==============================] - 0s 239us/sample - loss: 1.1843 - acc: 0.1250
Epoch 93/100
8/8 [==============================] - 0s 286us/sample - loss: 1.1826 - acc: 0.1250
Epoch 94/100
8/8 [==============================] - 0s 189us/sample - loss: 1.1809 - acc: 0.1250
Epoch 95/100
8/8 [==============================] - 0s 243us/sample - loss: 1.1792 - acc: 0.1250
Epoch 96/100
8/8 [==============================] - 0s 186us/sample - loss: 1.1775 - acc: 0.1250
Epoch 97/100
8/8 [==============================] - 0s 256us/sample - loss: 1.1758 - acc: 0.1250
Epoch 98/100
8/8 [==============================] - 0s 234us/sample - loss: 1.1742 - acc: 0.1250
Epoch 99/100
8/8 [==============================] - 0s 551us/sample - loss: 1.1725 - acc: 0.1250
Epoch 100/100
8/8 [==============================] - 0s 300us/sample - loss: 1.1708 - acc: 0.1250
[array([[ 0.37017596,  1.0379488 ,  0.740579  ],
       [-0.8259191 , -0.38164407, -0.61800957]], dtype=float32), array([0., 0., 0.], dtype=float32)]
```

FIGURE *13 – Modèle avec Poids*

4.2 Perceptron simple, multi-classes

On utilise plusieurs neurones dans la couche de sortie. On implémente des réseaux de neurones multicouches en utilisant Keras pour classifier des ensembles :

- Letter (.trn pour l'apprentissage, .tst pour le test)
- Iris (.trn pour l'apprentissage, .tst pour le test)
- Optics (.trn pour l'apprentissage, .tst pour le test)

```
In [11]: runfile('/home/azaria/Documents/tp_apprentissage/
tp_apprentissage_complet/exo5/tp2_No4_exo5.py', wdir='/home/azaria/Documents/
tp_apprentissage/tp_apprentissage_complet/exo5')
Epoch 1/20
6665/6665 [==============================] - 1s 164us/step - loss: 0.1403 -
acc: 0.9615
Epoch 2/20
6665/6665 [==============================] - 1s 104us/step - loss: 0.1000 -
acc: 0.9620
Epoch 3/20
6665/6665 [==============================] - 1s 104us/step - loss: 0.0899 -
acc: 0.9636
Epoch 4/20
6665/6665 [==============================] - 1s 105us/step - loss: 0.0854 -
acc: 0.9642
Epoch 5/20
6665/6665 [==============================] - 1s 104us/step - loss: 0.0824 -
IPython console    History log
```

Figure *14 – Modèle avec Letter1*

La figure 14 utilise 6 neurones dans la première couche cachée avec une 20 itération pour avoir le résultat Dans la suite de cette figure 15 , on voit la fonction de perte et la précision obtenue à 97%.

```
acc: 0.9729
Epoch 20/20
6665/6665 [==============================] - 1s 110us/step - loss: 0.0613 -
acc: 0.9734
[[False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [ True False False ... False False False]]
[[273   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0]
 [  0 240   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0]
 [  0   0 137  89   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0]
 [  0   0   0 277   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figure *15 – Modèle avec Letter2*

D'où pour ce programme par rapport à l'énoncer on a 4 couches entièrement connectées de « Dense » de « keras » pour créer notre réseau de neurone multicouche. Donc la première couche est définie avec dimension de l'entrée du réseau « input_dim ». Ensuite nous précisons le nombre d'unité de neurones cachés de chacune des couches de notre réseau de neurones à convolution (RNC).

Passons maintenant à notre expérimentation avec un autre jeu de donnée iris comme nous le montre la figure 16.

```
model = Sequential()
model.add(Dense(units = 6, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 6, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 6, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 3, kernel_initializer = 'uniform' , activation

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
model.fit(x, encode_y, batch_size = 10, epochs = 20)

y_predict = model.predict(x_test)
y_predict = (y_predict >= 0.4)
print(y_predict)

con_m = confusion_matrix(encode_ytest.argmax(axis=1), y_predict.argmax(
```

```
Epoch 14/20
99/99 [==============================] - 0s 182us/step - loss: 0.5881 - acc: 0.6667
Epoch 15/20
99/99 [==============================] - 0s 186us/step - loss: 0.5740 - acc: 0.6667
Epoch 16/20
99/99 [==============================] - 0s 164us/step - loss: 0.5594 - acc: 0.6667
Epoch 17/20
99/99 [==============================] - 0s 190us/step - loss: 0.5450 - acc: 0.7037
Epoch 18/20
99/99 [==============================] - 0s 166us/step - loss: 0.5313 - acc: 0.7677
Epoch 19/20
99/99 [==============================] - 0s 168us/step - loss: 0.5195 - acc: 0.7576
Epoch 20/20
99/99 [==============================] - 0s 150us/step - loss: 0.5101 - acc: 0.7172
```

Figure *16 – Modèle avec iris1*

Nous constatons que notre modèle n'a pas bien appris avec le 6 neurones cachés à la première couche car sa précision est faible et ce qui entraîne une fausse information sur la prédiction de résultat de certains individus. Nous expérimentons encore avec changement de valeur de nombre de neurones qui correspondent au nombre de classe de sortie comme le montre la figure 17

```
model = Sequential()
model.add(Dense(units = 32, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 24, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 6, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 3, kernel_initializer = 'uniform' , activation

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
model.fit(x, encode_y, batch_size = 16, epochs = 30)

y_predict = model.predict(x_test)
y_predict = (y_predict >= 0.4)
print(y_predict)

con_m = confusion_matrix(encode_ytest.argmax(axis=1), y_predict.argmax(
print(con_m)
```

```
99/99 [==============================] - 0s 176us/step - loss: 0.2602 - acc: 0.8687
Epoch 24/30
99/99 [==============================] - 0s 145us/step - loss: 0.2417 - acc: 0.8721
Epoch 25/30
99/99 [==============================] - 0s 140us/step - loss: 0.2217 - acc: 0.9259
Epoch 26/30
99/99 [==============================] - 0s 110us/step - loss: 0.2025 - acc: 0.9798
Epoch 27/30
99/99 [==============================] - 0s 134us/step - loss: 0.1829 - acc: 0.9899
Epoch 28/30
99/99 [==============================] - 0s 124us/step - loss: 0.1674 - acc: 1.0000
Epoch 29/30
99/99 [==============================] - 0s 120us/step - loss: 0.1483 - acc: 0.9933
Epoch 30/30
99/99 [==============================] - 0s 152us/step - loss: 0.1287 - acc: 1.0000
```

Figure *17 – Modèle d'apprentissage avec iris*

On a maintenant une précision de 100% et une prédiction correcte comme le montre la figure 18 avec sa matrice de confusion

```
 [False False  True]
 [False False  True]
 [ True False False]
 [False  True False]
 [ True False False]
 [False  True False]
 [False  True False]
 [False False  True]
 [False  True False]
 [False False  True]
 [False False  True]
 [False False  True]
 [False False  True]
 [ True False False]]
[[17  0  0]
 [ 0 14  0]
 [ 0  0 18]]
```

Figure *18 – Matrice de confusion Iris*

Nous continuons notre expérimentation avec un autre jeu de donnée Optics. La figure 19 représente les différentes informations appliquées pour avoir le résultat avec une précision de 86%.

```
model = Sequential()
model.add(Dense(units = 16, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 8, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 6, kernel_initializer = 'uniform' , activation
model.add(Dense(units = 4, kernel_initializer = 'uniform' , activation

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics
model.fit(x, encode_y, batch_size = 16, epochs = 30)

y_predict = model.predict(x_test)
y_predict = (y_predict >= 0.4)
print(y_predict)

con_m = confusion_matrix(encode_ytest.argmax(axis=1), y_predict.argmax(
print(con_m)
```

```
Epoch 23/30
3041/3041 [==============================] - 0s 129us/step - loss: 0.3512 - acc: 0.8580
Epoch 24/30
3041/3041 [==============================] - 0s 130us/step - loss: 0.3490 - acc: 0.8577
Epoch 25/30
3041/3041 [==============================] - 0s 158us/step - loss: 0.3470 - acc: 0.8585
Epoch 26/30
3041/3041 [==============================] - 0s 132us/step - loss: 0.3457 - acc: 0.8604
Epoch 27/30
3041/3041 [==============================] - 0s 131us/step - loss: 0.3441 - acc: 0.8592
Epoch 28/30
3041/3041 [==============================] - 0s 130us/step - loss: 0.3410 - acc: 0.8601
Epoch 29/30
3041/3041 [==============================] - 0s 122us/step - loss: 0.3399 - acc: 0.8605
Epoch 30/30
3041/3041 [==============================] - 0s 124us/step - loss: 0.3398 - acc: 0.8607
```

Figure *19 – Modèle d'apprentissage avec Optics*

La figure 20 représente la matrice de confusion de la figure 19.

```
[[ True False False False]
 [ True False False False]
 [ True False False False]
 ...
 [ True False False False]
 [False False False False]
 [ True False False False]]
[[513   0   0   0]
 [110   0   0   0]
 [ 39   0   0   0]
 [119   0   0   0]]
```

FIGURE *20 – Matrice de confusion*

On constate que la prédiction est bonne y compris sa matrice de confusion.

5 Conclusion et Perspective

En guise de conclusion, ce travail nous a permis d'avoir une idée générale sur les Réseaux de Neurones en l'occurrence les perceptrons simple et multiple, les Arbres de décision et les Séparateurs a Vaste Marge (SVM) afin de mettre en pratique ces différents algorithmes en utilisant ces differentes bibliothèques en python.

Les résultats obtenus au cours de nos expérimentations étaient valides et nous permettent de constater que certains algorithmes sont plus performants que les autres en termes d'apprentissage et que les paramètres permettant de réaliser ces apprentissages influencent les résultats.

Références

[1] Les méthodes ensembliste, http ://blog.octo.com/les-methodes-ensemblistes-pour algorithmes-de-machine-learning/.

[2] https ://www.tensorflow.org/tutorials, Mars 2019

[3] Andreas C. Mueller and Sarah, Guido Introduction to Machine Learning with Python

[4] Aurélien Géron, Hands-On Machine Learning with Scikit-Learn and TensorFlow, Concepts, Tools, and Techniques to Build Intelligent Systems.