Université des Antilles

LAMIA
Laboratoire de Mathématiques Informatique et Applications

Tutoriel TensorFlow

IKNI Layachi
PAGÉ Vincent

1er janvier 2018

Table des matières

Chapitre 1

Introduction

1.1 Installation

TensorFlow est une librairie de calcul dédiée à l'apprentissage automatique. On peut l'utiliser avec python, java, C,.... Dans notre cas, nous utiliserons python.

Pour l'installation, nous avons suivi les instructions du tutoriel officiel qui se trouve ici, sans difficultés :

https ://www.tensorflow.org/install/

A noter pour les installations de TensorFlow avec virtualEnv (sous Linux), il nous semble raisonnable d'avoir un répertoire contenant les environnements virtuels, dont celui correspondant a TensorFlow (disons `~/VirtualEnvs/TensorFlow`) et un répertoire contenant les sources du projet (disons `~/DNN`).

1.2 Présentation TensorFlow et TensorBoard

1.2.1 Premiers concepts de TensorFlow

Tout d'abord, on peut utiliser TensorFlow de deux façons :

— En utilisant des algorithmes précodés, utilisant la classe **Estimator**

— En maitrisant chaque étape du calcul.

Dans le second cas, il est nécessaire de comprendre que TensorFlow s'appuie sur des concepts de programmation très différents d'une programmation standard python.

Pour bien comprendre ces concepts, prenons un exemple : On veut que notre programme prenne une valeur réelle (x), calcule une valeur $y = W * x + b$ avec W et b des valeurs réelles que notre programme sera appelé à modifier plus tard.

Le code correspondant en python est le suivant

```
x = 2
W = 0.3
b = -0.3
y = W*x+b
print(y)
```

La sortie de ce programme serait :

```
0.3
```

ici, W, b , x et y sont des variables du programmes. Néanmoins, dans le contexte de notre programme, elles jouent des rôles très différents :

— x est une entrée

— W et b sont des valeurs modifiables

— y est calculé à partir de x, W et b

La programmation en TensorFlow, met en place cette différence.

— x sera appelé un **placeholder** , (en deux mots : une variable dont on promet qu'on lui donnera une valeur au moment du run)

— W et b seront définis comme des variables

— y sera défini implicitement par l'équation de calcul

Notons qu'il existe aussi la notion de constante, non présentée ici, mais facile a appréhender.
Le code correspondant en TensorFlow est le suivant :

```
import tensorflow as tf

x = tf.placeholder(tf.float32)

W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-.3], dtype=tf.float32)

y = W*x + b

print(y)
```

La sortie de ce programme est alors surprenante :

```
Tensor("add:0", dtype=float32)
```

De fait, nous n'avons pas calculé la valeur de y.
En fait notre programme ne manipule pas des variables au sens traditionnel, mais explique les dépendances entres les différents éléments de notre programme (ce sont des nœuds du graphes de calcul). TensorFlow s'appuie sur ce graphe de calcul, sur lequel nous reviendrons plus tard pour comprendre son intérêt.

Le graphe correspondant est représenté ci-dessous pour information.

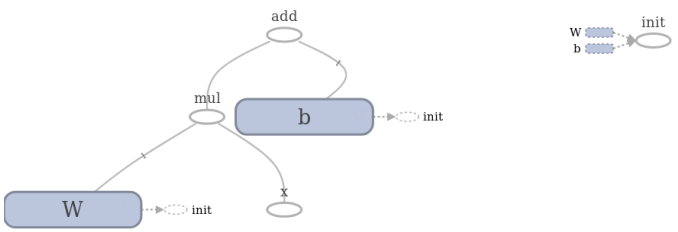

FIGURE 1.1 – Graphe de calcul simple

On retrouve dans ce graphes la présence des deux variables (W et b), le placeholder x, et un noeud *add* dont la sortie correspond à y.

Par ailleurs, les variables au sens TensorFlow sont des nœuds de calculs qui peuvent être modifiés. Elle ne sont pas initialisées par leur déclaration. Il faudra explicitement demander leur initialisation pour qu'elles agissent comme on s'y attend. Il s'agit maintenant pour que notre programme calcule bien la valeur voulue de :

— construire le graphe de calcul a partir des informations précédentes.

— initialiser les variables W et b

— lancer le calcul de y avec une valeur choisie pour x...

Ajoutons les codes suivants à notre programme

```
sess = tf.Session()
```

Le code ci-dessus construit le graphe.

```
init = tf.global_variables_initializer()
sess.run(init)
```

Le code ci-dessus initialise toutes les variables du programme (W et b). En fait, ce code construit un noeud de calcul correspondant a l'initialisation (première ligne) et lance le calcul correspondant (deuxième ligne).

```
resu = sess.run(y, {x:2})
print(resu)
```

Ce code lance le calcul de y, en prenant soin de placer la valeur 2 dans le placeholder x et afficher le résultat attendu

```
[0.3]
```

Pour comprendre l'intérêt de ces concepts de graphe de calcul, ajoutons à la fin de notre programme existant le code suivant :

```
resu = sess.run(y, {x:[1, 2, 3]})
print(resu)
```

Cette fois ci, les sorties sont :

```
[0.3]
[0.  0.3 0.6]
```

Nous avons en fait lancé deux runs (deux calculs de y). La première fois, x est un réel, la seconde fois x est un tableau de réels. Dans le second cas, pour chaque valeur de x, une valeur est calculée pour y. Notre programme a donc mis en place une procédure de calcul (le graphe de calcul) que l'on peut utiliser de multiples fois, avec différentes valeurs d'entrées (qui de plus prennent des formes différentes).

Une grande partie de la force de TensorFlow tient dans ces notions.

Voici donc le code du programme complet. Ce code est contenu dans le fichier : `DNN/Documentation/TutosPython/PremiersCalculs/calculTensorFlow.py`

```
import tensorflow as tf

x = tf.placeholder(tf.float32, name="x")

W = tf.Variable([.3], dtype=tf.float32, name="W")
b = tf.Variable([-.3], dtype=tf.float32, name="b")

y = W*x + b

print(y)

sess = tf.Session()

init = tf.global_variables_initializer()
sess.run(init)

resu = sess.run(y, {x:2})
print(resu)

resu = sess.run(y, {x:[1, 2, 3]})
print(resu)
```

../../TutosPython/PremiersCalculs/calculTensorFlow.py

Formalisation des concepts

Ceci tient en quelques mots :

— TensorFlow s'appuie sur un **graphe de calcul**.

— Les nœuds de ce graphe sont des **opérations**.

— Sur les arêtes de ce graphe circulent des **tenseurs**. Ces tenseurs sont les sorties des nœuds du graphe.

— Faire des calculs avec TensorFlow, c'est lancer un run en demandant le calcul d'une de ces sorties.

1.2.2 Premiers pas avec Tensorboard

TensorBoard est l'outil de visualisation associé à TensorFlow. Il permet de visualiser le graphe de calcul, mais aussi des valeurs importantes retenues lors des calculs exécutés sur ce graphe de calcul.

Pour sélectionner les informations a visualiser, nous l'indiquerons a notre programme TensorFlow. Le programme sauvegardera ces informations dans un répertoire spécifique.

On pourra alors lancer l'exécutable TensorBoard qui va analyser ce répertoire, créer un serveur web local que l'on pourra consulter pour visualiser nos informations... Voyons comment tout ceci se fait.

Ajout de code dans le programme TensorFlow (a la fin du programme précédent) :

```
pathLog="./SaveHere/";
writer = tf.summary.FileWriter(pathLog, sess.graph)
writer.close()
```

Ici, on choisit le répertoire (répertoire de Log) dans lequel seront stockées les informations importantes, et on crée un objet permettant d'écrire les informations de notre programme sur le disque. Ici, nous ne sauvons que le graphe de calcul. Enfin, on ferme cet objet.

Le code du programme complet est contenu dans le fichier :
`DNN/Documentation/TutosPython/PremiersCalculs/calculTensorFlowWithTensorBoard.py`

```
import tensorflow as tf

x = tf.placeholder(tf.float32, name="x")

W = tf.Variable([.3], dtype=tf.float32, name="W")
b = tf.Variable([-.3], dtype=tf.float32, name="b")

y = W*x + b

print(y)

sess = tf.Session()

init = tf.global_variables_initializer()
sess.run(init)

resu = sess.run(y, {x:2})
print(resu)

resu = sess.run(y, {x:[1, 2, 3]})
print(resu)

pathLog="./SaveHere/";
writer = tf.summary.FileWriter(pathLog, sess.graph)
writer.close()
```

../../TutosPython/PremiersCalculs/calculTensorFlowWithTensorBoard.py

On lance notre programme TensorFlow

```
python .\calculTensorFlowWithTensorBoard.py
```

On lance ensuite TensorBoard sur le répertoire de Log avec la ligne suivante :

```
tensorboard --logdir=./SaveHere
```

L'exécution donne le message suivant :

```
Starting TensorBoard b'41' on port 6006
(You can navigate to http://127.0.1.1:6006)
```

Enfin, on ouvre un navigateur dans lequel on indique l'URL de consultation indiquée par la sortie précédente.

Voici une capture de la visualisation obtenue :

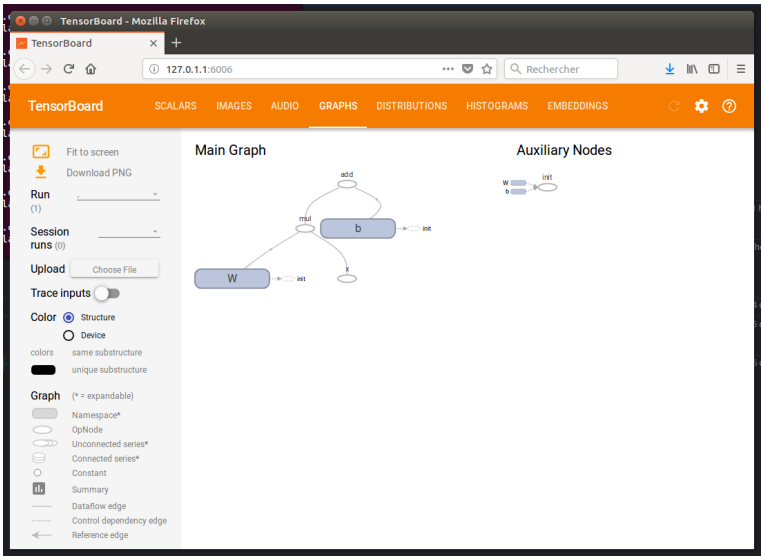

FIGURE 1.2 – La fenêtre de visualisation de TensorBoard

On retrouve ici le graphe précédent. Notez le sous graphe en haut a droite (noeud init) correspondant à l'initialisation des variables dont nous avons parlé précédemment. On verra plus loin que TensorBoard nous permet aussi de visualiser l'évolution de nos apprentissages...

Chapitre 2

Première Classification : Base IRIS

Sur cette base comme pour les autres, nous allons considérer deux cas :

— on crée le classifieur manuellement (boite blanche)

— on utilise un classifieur prédéfini (boite noire)

Pour chaque classifieur, nous sauverons le modèle après l'apprentissage, et le rechargerons pour les prédictions.

2.1 La Base Iris

l' ensemble de données Iris est utilisé pour prédire les espèces de fleurs basées sur la géométrie des sépales / pétales. Pour cet exemple les données Iris ont été randomisées et divisées en deux CSV distincts :

— Un ensemble d'apprentissage de 120 échantillons (`iris_training.csv`)

— Un ensemble de test de 30 échantillons (`iris_test.csv`).

Cette base a 3 classes possibles et chaque exemple présente 4 caractéristiques.

2.2 Réseau monocouche boite blanche

Le code complet de ce programme sera donné en fin de section. Voyons quelques grandes étapes.

2.2.1 lecture des données

Supposons que les données soient déja présentes sur le disque, on les lira comme suit, par exemple pour la base d'apprentissage.

```
# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename="iris_training.csv",
  target_dtype=np.int,
  features_dtype=np.float32)
```

L'objet `training_set` a un champ `data` qui contient les paramètres des exemples et un champ `target` qui contient les labels des classes.

2.2.2 Construction du réseau

Les entrées

On crée ensuite les placeholders pour entrer ces données : X est bien de taille 4.

```
with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 4], name = "X")

with tf.name_scope('Y_True'):
    # sorties voulues
    y_int = tf.placeholder(tf.uint8, [None], name = "Y_int")
    y_ = tf.one_hot(y_int, depth=3, name = "Y_True")
```

A noter que l'on crée ici un noeud `y_` qui transforme le label (entier) d'une classe en un **hot vector**. Par exemple, le label 0 parmi 3 classes possibles se transforme en hot vecteur : `[1,0,0]`

Le modèle

Définissons notre modèle : un réseau monocouche linéaire. Nous avons 3 neurones de sortie (1 par classe). Chaque neurone a 4 entrées (les 4 paramètres). Nous avons 4 poids et 1 biais par neurone. Le score calculé pour chaque classe est classique.

```
# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([4, 3]),name ="W")

with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([3]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b
```

Les lignes de type `with tf.name_scope("Score"):` servent juste à regrouper certains nœuds pour la visualisation avec TensorBoard.

optimisation du modèle

Pour trouver les meilleurs poids (W, b) du réseau, il nous faut : un critère à minimiser, et un algorithme de minimisation.

Le critère à minimiser sera pour nous **l'entropie croisée**. Pour le calculer, nous transformons les scores de chaque sortie en une probabilité (fonction `softmax`). On veut ensuite calculer une distance entre ce vecteur de probabilité et la distribution voulue (le hot vector d'entrée). L'entropie croisée fera office de distance.

```
with tf.name_scope('softmax'):
    y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))
```

Pour savoir quelle est la classe prédite il faut regarder quelle sortie a donné le plus fort score :

```
classe = tf.argmax(y,1)
```

Pour mesurer la précision (le taux de reconnaissance), il faut vérifier si la classe retenue pour un exemple est bien la classe donnée par la base. On calcule ensuite le taux moyen de succès sur un ensemble d'exemples :

```
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

On choisit enfin une méthode d'optimisation. Ici, une descente de gradient appliquée à notre entropie croisée.

```
# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Cette descente de gradient définit un nœud de calcul `train_step` que l'on pourra appeler lors du run.

2.2.3 lancement des calculs

On initialise tout d'abord les variables du réseau.

```
init = tf.global_variables_initializer();
sess.run(init);
```

Apprentissage : On lance le run sur le noeud de calcul `train_step`.

```
for i in range(1000):
  sess.run(train_step, feed_dict={x: training_set.data, y_int: training_set.target})
```

Evaluons notre apprentissage sur la base d'apprentissage : On lance le run sur le noeud de calcul correspondant à la précision.

```
print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: training_set.
    data, y_int: training_set.target}))
```

Voyons une prédiction : on lance le run sur le noeud de calcul de classe, avec des exemples inconnus.

```
new_samples = np.array(
  [[6.9, 3.2, 4.5, 1.5],
   [4.8, 3.1, 5.0, 1.7]], dtype=np.float32)

print("classe ", sess.run(classe, {x: new_samples}))
```

A ce stade, nous avons un classifieur fonctionnel. Ajoutons quelques améliorations pour la visualisation avec TensorBoard.

2.2.4 TensorBoard : Evolution des performances

Nous allons modifier notre code pour disposer de graphiques d'évolution de la précision et de l'entropie croisée au cours de l'apprentissage.

Commençons par choisir un répertoire, et ouvrons un `FileWriter` pour écrire dedans

```
visuPath = './VisuMonoCouche'
writer = tf.summary.FileWriter(visuPath, sess.graph)
```

Signalons ensuite que nous voulons suivre l'entropie croisée et la précision. Ces infos seront fusionnées pour le `FileWriter` en un nœud `merged`.

```
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

merged = tf.summary.merge_all()
```

L'apprentissage doit etre un peu modifié : Quand on lance un run d'apprentissage, on calcule aussi le noeud `merged`. Sa sortie est récupérée dans la variable `summary`. Celle ci est passée au `FileWriter` qui gère la sauvegarde.

```
for i in range(1000):
  summary, _ = sess.run([merged,train_step], feed_dict={x: training_set.data, y_int:
    training_set.target})

  writer.add_summary(summary, i)
```

Ce code permet d'obtenir des visualisations telles que celle ci :

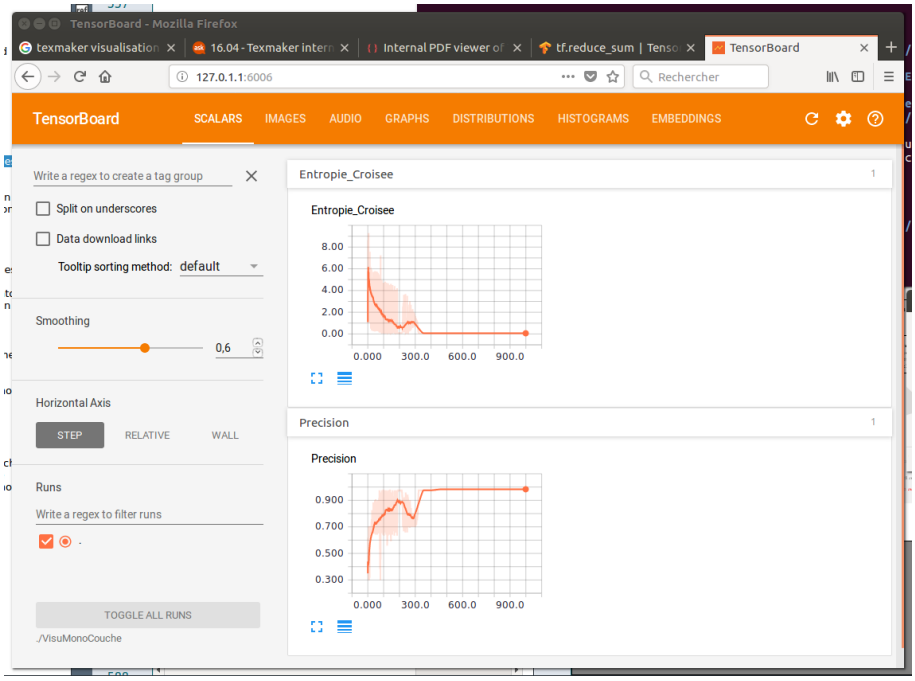

FIGURE 2.1 – Évolution des performances en apprentissage

2.2.5 Code du programme complet

On peut améliorer un peu notre programme en téléchargeant la base IRIS si on ne l'a pas sur le disque, et effacer le répertoire de visualisation pour éviter qu'il ne se remplisse de données, ce qui donnerait le programme complet suivant contenu dans le fichier :
`DNN/Documentation/TutosPython/Iris/irisMonocoucheComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen

import numpy as np
import shutil


import tensorflow as tf

# Data sets
IRIS_TRAINING = "IrisDatabase/iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "IrisDatabase/iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

# If the training and test sets aren't stored locally, download them.
if not os.path.exists(IRIS_TRAINING):
    raw = urlopen(IRIS_TRAINING_URL).read()
    with open(IRIS_TRAINING, "wb") as f:
        f.write(raw)

if not os.path.exists(IRIS_TEST):
    raw = urlopen(IRIS_TEST_URL).read()
    with open(IRIS_TEST, "wb") as f:
        f.write(raw)

#mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)


# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TRAINING,
  target_dtype=np.int,
  features_dtype=np.float32)

test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TEST,
  target_dtype=np.int,
  features_dtype=np.float32)




with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 4], name = "X")

with tf.name_scope('Y_True'):
    # sorties voulues
```

```
    y_int = tf.placeholder(tf.uint8, [None], name = "Y_int")
    y_ = tf.one_hot(y_int, depth=3, name = "Y_True")



# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([4, 3]),name ="W")

with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([3]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b



# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
    y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#   cross_entropy = tf.reduce_mean(
#       tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

classe = tf.argmax(y,1)

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.Session()


# Configuration de TensorBoard
# If the visu dir exists, we delete it.
# to avoid accidental multiple trainings visualisation
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)


writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()
```

```
init = tf.global_variables_initializer();
sess.run(init);


for i in range(1000):
  summary, _ = sess.run([merged,train_step], feed_dict={x: training_set.data, y_int:
    training_set.target})
  #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
     batch_ys}))

  writer.add_summary(summary, i)

print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: training_set.
    data, y_int: training_set.target}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: test_set.data,
    y_int: test_set.target}))

# Classify two new flower samples.
new_samples = np.array(
  [[6.9, 3.2, 4.5, 1.5],
   [4.8, 3.1, 5.0, 1.7]], dtype=np.float32)

print("classe ", sess.run(classe, {x: new_samples}))

writer.close()
```

../../TutosPython/Iris/irisMonocoucheComplet.py

2.2.6 Sauvegarde et Chargement d'un réseau

Il s'agit maintenant de séparer notre programme en deux :

— Un programme qui apprend, évalue les performances et sauve le réseau

— Un programme qui charge un réseau entraîné, et fait des prédictions.

Sauvegarde

La sauvegarde sera prise en charge par l'objet `train.Saver`, qui va sauver le graphe, les variables du graphes (et d'autres choses plus nébuleuses pour les rédacteurs) dans un répertoire. L'essentiel du code qui fait cela est le suivant.

```
saver = tf.train.Saver()

savePath = './SavedNetworks/myMonoCouchemodel.ckpt'

saver.save(sess, savePathFull)
```

Ceci va générer :

— un fichier `myMonoCouchemodel.ckpt.meta` qui contient l'architecture du réseau.

— un fichier `myMonoCouchemodel.ckpt.data-00000-of-00001` qui contient les valeurs des variables du réseau à l'instant de sauvegarde.

— un fichier `myMonoCouchemodel.ckpt.index` qui, je crois, contient le nom des noeuds du réseau.

— un fichier `checkpoint` utile mais sans qu'on sache trop pourquoi.

Chargement

le chargement n'est pas trop compliqué non plus :

— on charge le fichier d'architecture, ce qui crée le graphe.

— on recharge l'état du réseau dans le graphe.

— on récupère les noeuds du graphe qui nous intéressent.

```
sess = tf.Session()

new_saver = tf.train.import_meta_graph('./SavedNetworks/myMonoCouchemodel.ckpt')

new_saver.restore(sess, tf.train.latest_checkpoint('./SavedNetworks/'))
```

A ce stade, nous disposons d'un réseau pré-entraîné. Notre programme doit donc lui injecter des données et récupérer certaines sorties. Commencons par obtenir du graphe des pointeurs sur tous ces nœuds d'intérêt.

Lorsque le réseau a été construit lors de l'apprentissage, nous avons crée par exemple le nœud qui calcule la classe comme suit :

```
with tf.name_scope('Classe'):
    classe = tf.argmax(y,1, name="classe")
```

Le nom complet de ce noeud dans le graphe est "Classe/classe :0". "Classe" pour le nameScope s'il y en a un. "classe" correspond le nom du nœud. " :0" est un indice systématiquement ajouté et auto-incrémenté pour éviter les déclarations multiples accidentelles.

Pour une prédiction, nous devons obtenir des pointeurs sur le placeholder x et sur le nœud des classes :

```
graph = tf.get_default_graph()

x = graph.get_tensor_by_name("X/X:0")

classe = graph.get_tensor_by_name("Classe/classe:0")
```

Il suffit ensuite de lui fournir normalement les exemples pour obtenir les classes.

```
new_samples = np.array(
    [[5.2, 1.2, 4.5, 1.5],
    [4.8, 6.4, 5.0, 1.7],
    [3.5, 3.2, 2.5, 8.5]], dtype=np.float32)

predictions = sess.run(classe, {x: new_samples})
```

2.2.7 Programmes de sauvegarde et prédiction complets

Dans ces programmes, nous avons ajouté quelques lignes pour paramétrer proprement les noms des fichiers et répertoires de sauvegarde et une procédure qui efface les répertoires de sauvegarde des modèles s'ils existent.

Voici le code du programme qui réalise l'apprentissage, contenu dans le fichier : `DNN/Documentation/TutosPython/Iris/irisMonocoucheTrain.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen

import numpy as np
import shutil

import tensorflow as tf

# Data sets
IRIS_TRAINING = "IrisDatabase/iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "IrisDatabase/iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

# If the training and test sets aren't stored locally, download them.
if not os.path.exists(IRIS_TRAINING):
    raw = urlopen(IRIS_TRAINING_URL).read()
    with open(IRIS_TRAINING, "wb") as f:
        f.write(raw)

if not os.path.exists(IRIS_TEST):
    raw = urlopen(IRIS_TEST_URL).read()
    with open(IRIS_TEST, "wb") as f:
        f.write(raw)


# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TRAINING,
  target_dtype=np.int,
  features_dtype=np.float32)

test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TEST,
  target_dtype=np.int,
  features_dtype=np.float32)


with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 4], name = "X")

with tf.name_scope('Y_True'):
    # sorties voulues
    y_int = tf.placeholder(tf.uint8, [None], name = "Y_int")
    y_ = tf.one_hot(y_int, depth=3, name = "Y_True")



# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([4, 3]),name ="W")
```

```
with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([3]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b



# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
    y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#    cross_entropy = tf.reduce_mean(
#        tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name="accuracy
    ")

with tf.name_scope('Classe'):
    classe = tf.argmax(y,1, name="classe")

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.Session()



# Configuration de TensorBoard
# If the visu dir exists, we delete it.
# to avoid accidental multiple trainings visualisation
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)


writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()



init = tf.global_variables_initializer();
sess.run(init);


for i in range(1000):
```

```
    summary, _ = sess.run([merged,train_step], feed_dict={x: training_set.data, y_int:
      training_set.target})
    #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
       batch_ys}))

    writer.add_summary(summary, i)


#Create a saver object which will save all the variables
saver = tf.train.Saver()

# sauvegarde en fin d'apprentissage
savePath = 'SavedNetworks/'
modelName = 'myMonoCouchemodel.ckpt'
if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

savePathFull = os.path.join(savePath, modelName)

saver.save(sess, savePathFull)


print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: training_set.
    data, y_int: training_set.target}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: test_set.data,
    y_int: test_set.target}))
```

../../TutosPython/Iris/irisMonocoucheTrain.py

Voici le code du programme qui réalise les prédictions, contenu dans le fichier : `DNN/Documentation/TutosPython/Iris/irisMonocouchePredict.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import numpy as np

import tensorflow as tf


# Chargement du réseau
savePath = 'SavedNetworks'
modelName = 'myMonoCouchemodel.ckpt'
savePathFull = os.path.join(savePath, modelName)
print ("ModelFilename ", repr(savePathFull))

metagraphFilename = savePathFull+'.meta'

print ("restoring graph ", repr(metagraphFilename))

with tf.Session() as sess:

  #print ("restoring graph ", metagraphFilename)
  new_saver = tf.train.import_meta_graph(metagraphFilename)


  print ("restoring variables from latest checkpoint in", savePath)
  new_saver.restore(sess, tf.train.latest_checkpoint(savePath))
```

```
graph = tf.get_default_graph()
x = graph.get_tensor_by_name("X/X:0")

classe = graph.get_tensor_by_name("Classe/classe:0")

new_samples = np.array(
  [[5.2, 1.2, 4.5, 1.5],
  [4.8, 6.4, 5.0, 1.7],
  [3.5, 3.2, 2.5, 8.5]], dtype=np.float32)

predictions = sess.run(classe, {x: new_samples})

dicoClasses = ['setosa', 'versicolor', 'virginica']

for p in predictions :
  print ("je pense que c'est : ",dicoClasses[p])
```

../../TutosPython/Iris/irisMonocouchePredict.py

2.2.8 Quelques remarques sur l'implémentation

Le lecteur soucieux de faire fonctionner ses programmes pourra sans risque sauter cette section pour revenir dessus quand il aura mieux pris en main TensorFlow.

TensorFlow travaille avec des tenseurs (des tableaux multi-dimensionnels). Il est intéressant de comprendre quelles tailles ont ces tenseurs dans notre programme.

Dans tous nos programmes, les données insérées dans notre réseau seront de type `NdArray` de la librairie `numpy`. Leur taille doit correspondre aux placeholders du réseau. Par exemple :

— `training_set.data` est un NdArray de dimension 2, de taille (120,4)

— `training_set.data` est un NdArray de dimension 1, de taille (120)

Les placeholders ont les caractéristiques suivantes

— `x` a pour taille (None,4) : il accepte autant d'exemples que fournis, tous de 4 caractéristiques, chacune en float32.

— `Y_` (les labels) a pour taille (None) : il accepte autant de labels que fournis, tous d'une seule valeur entiere.

Il faut bien comprendre que le calcul de toutes les sorties pour toutes les entrées se fait en une seule opération. Voyons donc le détail de ceci.

la matrice des poids W est un tenseur de dimension 2, de taille (4,3).

le réseau calcule $tf.matmul(x, W)$: Le résultat de cette opération est, dans le cas d'un seul exemple, un tenseur de dimension 2, de taille (1, 3)

On lui ajoute le biais b, qui est un tenseur de dimension 1, de taille 3. Le résultat est toujours un tenseur de dimension 1, de taille 3 représentant chaque le score en sortie du réseau pour l'exemple proposé.

Dans le cas ou l'on présente 120 exemples :

Le résultat de $tf.matmul(x, W)$ est un tenseur de dimension 2, de taille (120, 3).

On lui ajoute le tenseur de biais, de dimension 1, et de taille 3. La réussite opérationnelle de cette opération est intéressante [1]

A partir de ces scores, on calcule les probabilités y, qui sont toujours un tenseur d'ordre 2, de taille (120, 3)

`y_`, le hot vector correspondant aux labels est lui aussi un tenseur d'ordre 2, de taille (120,3).

1. Elle est due au concept de broadcasting. voir https ://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

La multiplication impliquée dans le calcul de l'entropie croisée est une multiplication point à point.

`tf.reduce_sum` doit calculer l'entropie pour un exemple. Il opère bien sur la direction 1 (il somme les 3 sorties). Sa sortie est un tenseur d'ordre 1 de taille (120).

`tf.reduce_mean` doit calculer la moyenne de ces entropies sur tous les exemples. Le tenseur qu'on lui fournit en entrée etant d'ordre 1, il n'y a pas besoin de spécifier de direction pour cette opération.Sa sortie est un tenseur d'ordre 1 de taille (1).

2.3 Réseau multi couche Boite Noire

Dans cette section, nous reprenons tout ce qui a été fait précédemment, en modifiant le modèle, et en utilisant un réseau pré-codé, intégré dans la classe Estimator.

Ce réseau est de type DNN, nous le créerons avec seulement 3 couches.

la classe Estimator dont hérite notre réseau encapsule la plupart des opérations et donne un code beaucoup plus classique, tout le graphe étant masqué.

2.3.1 lecture des données

Rien de changé, le lecteur peut se référer à la section 2.2.1.

2.3.2 Construction du réseau

```
feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
```

Ce code prépare une variable pour les entrées du modèle (le vecteur de caractéristiques ou **feature vector**) et spécifie que **feature vector** est numérique et de dimension 4 (longueur des sépales, largeurs des sépales, longueur des pétales, largeur des pétales).

```
classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                        hidden_units=[10, 20, 10],
                                        n_classes=3,
                                        model_dir="/VisuDnn")
```

Ce code crée un Réseau de Neurones Profond (Deep Neural Network ou **DNN**) que l'on utilisera pour classifier nos données. Pour le construire on lui passe les informations suivantes :

- la forme du **feature vector**.
- la topologie du réseau. Ici : trois couches cachées , contenant respectivement 10, 20 et 10 neurones.
- le nombre de classes attendues en sortie. Ici : Trois classes, représentant les trois espèces d'Iris.
- le répertoire dans lequel TensorFlow sauvegardera en particulier les données utilisées par TensorBoard pour la visualisation.

2.3.3 Lancement des calculs

La classe Estimator propose trois grandes actions lors d'un apprentissage :

- Train : on entraine le réseau
- Evaluate : on évalue le réseau sur une base

— Predict : on effectue des prédictions sur de nouveaux exemples.

Voyons par exemple comment lancer l'apprentissage. la classe Estimator. impose la structure de code suivante :

— on définit une fonction spécifiant :
  — le jeu de donnée utilisé (caractéristiques et labels).
  — le nombre d'**epochs** lors de l'apprentissage.
  — si les données doivent être présentées dans un ordre aléatoire
— on lance l'entrainement qui utilise cette fonction un certain nombre de fois.

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": np.array(training_set.data)},
      y=np.array(training_set.target),
      num_epochs=None,
      shuffle=True)

classifier.train(input_fn=train_input_fn, steps=2000)
```

On suit la même logique lorsqu'on veut évaluer le classifieur sur la base d'apprentissage, mais on lance cette fois la fonction d'évaluation du classifieur.

```
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": np.array(training_set.data)},
      y=np.array(training_set.target),
      num_epochs=1,
      shuffle=False)


accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("\nLearning Accuracy: {0:f}\n".format(accuracy_score))
```

Enfin, on évalue le classifieur sur la base de généralisation.

```
test_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": np.array(test_set.data)},
      y=np.array(test_set.target),
      num_epochs=1,
      shuffle=False)
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]
print("\nTest Accuracy: {0:f}\n".format(accuracy_score))
```

la sortie de ce programme donne quelque chose comme : soit une précision en généralisation de 0.97

```
Learning Accuracy: 1.000
Test Accuracy: 0.966667
```

Si l'on souhaite prédire les classes de nouveaux exemples, on peut procéder comme suit en lançant la fonction de prédiction du classifieur, dont on affichera les résultats.

```
new_samples = np.array(
      [[6.9, 3.2, 4.5, 1.5],
```

```
        [4.8, 3.1, 5.0, 1.7]], dtype=np.float32)

predict_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={"x": new_samples},
      num_epochs=1,
      shuffle=False)

predictions = classifier.predict(input_fn=predict_input_fn)

for p in predictions :
    class_id = p['class_ids'][0]
    probability = p['probabilities'][class_id]
    print ("je pense que c'est : ",dicoClasses[class_id], "avec une proba de ",
    probability )
```

La sortie du programme complet serait :

```
Learning Accuracy: 1.000000
Test Accuracy: 0.966667
classe  1
classe  2
```

2.3.4 TensorBoard : Evolution des performances

Cette fois ci, nous n'avons pas à spécifier quelles variables nous intéressent dans le réseau. Estimator procède a la sauvegarde de l'évolution des performances automatiquement.

Il suffit donc de lancer TensorBoard sur le répertoire que nous avons passé a Estimator lors de la création du modèle. Nous avions écrit :

```
classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                        hidden_units=[10, 20, 10],
                                        n_classes=3,
                                        model_dir="./VisuDnn")
```

on lancera donc dans un autre terminal la commande

```
tensorboard --logdir=./VisuDnn
```

Ci dessous, les figures respectives de l'évolution des paramètres du réseau pendant l'apprentissage et du graphe de calcul.

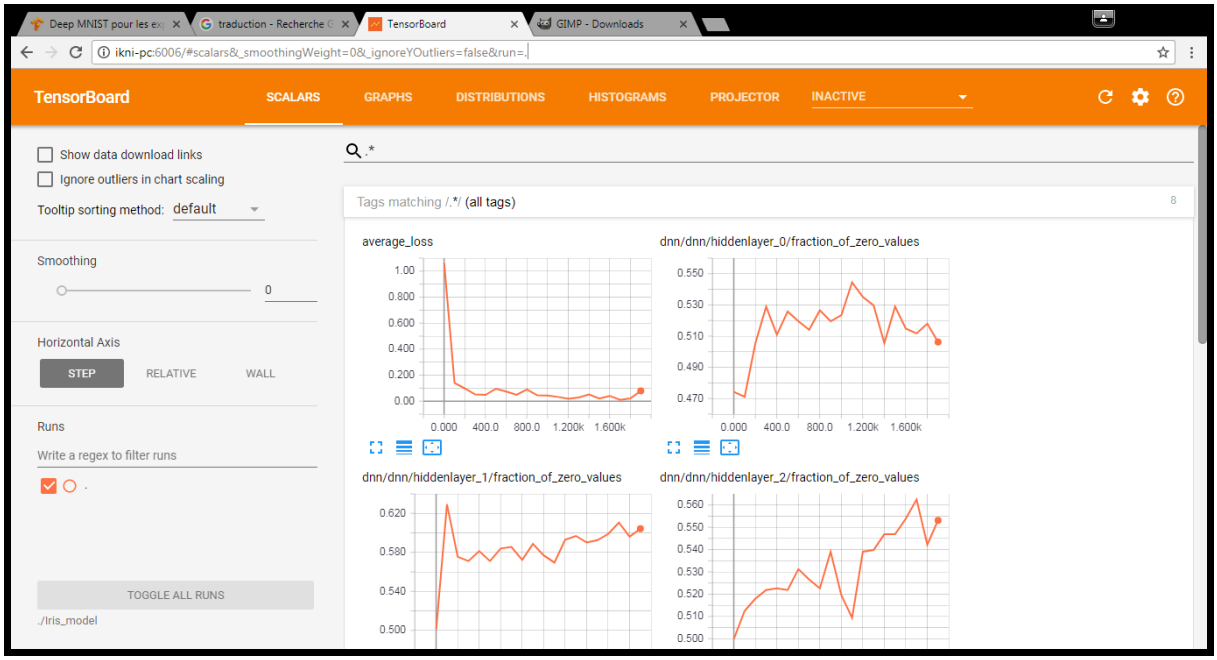

FIGURE 2.2 – Evolution des paramètres du réseau sur la base IRIS

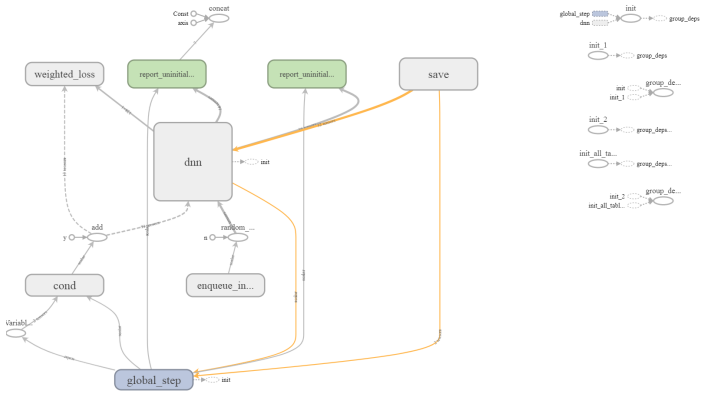

FIGURE 2.3 – Graphe de calcul du DNN utilisé sur la base IRIS

2.3.5 Code du programme complet

Notre programme est maintenant complet. Nous pouvons le trouver dans le fichier : `DNN/Documentation/TutosPython/Iris/irisDnnEstimatorComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen

import numpy as np
```

```
import shutil

import tensorflow as tf

# Data sets
IRIS_TRAINING = "IrisDatabase/iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "IrisDatabase/iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

# If the training and test sets aren't stored locally, download them.
if not os.path.exists(IRIS_TRAINING):
    raw = urlopen(IRIS_TRAINING_URL).read()
    with open(IRIS_TRAINING, "wb") as f:
        f.write(raw)

if not os.path.exists(IRIS_TEST):
    raw = urlopen(IRIS_TEST_URL).read()
    with open(IRIS_TEST, "wb") as f:
        f.write(raw)

# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TRAINING,
  target_dtype=np.int,
  features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TEST,
  target_dtype=np.int,
  features_dtype=np.float32)

# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]

# Build 3 layer DNN with 10, 20, 10 units respectively.

# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                        hidden_units=[10, 20, 10],
                                        n_classes=3,
                                        model_dir=visuPath)
# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(training_set.data)},
  y=np.array(training_set.target),
  num_epochs=None,
  shuffle=True)


# Train model.
classifier.train(input_fn=train_input_fn, steps=2000)


# Save Model
```

```
# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(training_set.data)},
  y=np.array(training_set.target),
  num_epochs=1,
  shuffle=False)


accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("\nLearning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(test_set.data)},
  y=np.array(test_set.target),
  num_epochs=1,
  shuffle=False)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("\nTest Accuracy: {0:f}\n".format(accuracy_score))



# Classify two new flower samples.
new_samples = np.array(
  [[6.9, 3.2, 4.5, 1.5],
   [4.8, 3.1, 5.0, 1.7]], dtype=np.float32)
predict_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": new_samples},
  num_epochs=1,
  shuffle=False)

dicoClasses = ['setosa', 'versicolor', 'virginica']
predictions = classifier.predict(input_fn=predict_input_fn)

for p in predictions :
    class_id = p['class_ids'][0]
    probability = p['probabilities'][class_id]
    print ("je pense que c'est : ",dicoClasses[class_id], "avec une proba de ",
    probability )
```

../../TutosPython/Iris/irisDnnEstimatorComplet.py

2.3.6 Sauvegarde et Chargement avec Estimator

Il s'agit maintenant de séparer notre programme en deux :

— Un programme qui apprend, évalue les performances et sauve le réseau

— Un programme qui charge un réseau entraîné, et fait des prédictions.

Nous ne disposons plus d'un graphe, ni même d'une session. La procédure est donc totalement différente de ce qui a été vu dans la section 2.2.6.

Sauvegarde

Les classifieurs de la classe Estimator savent sauvegarder leurs modèles grâce a la méthode `export_savedmodel`.

Cependant cette méthode suppose la définition d'un modèle de service des données. Le modèle qui suit est fonctionnel et a été reconstitué à l'aide de fragments trouvés sur le web, sans que l'on ne comprenne vraiment tout ce qui s'y passe.

```
def serving_input_receiver_fn():
  feature_spec = {'x': tf.FixedLenFeature([4],tf.float32)}
  serialized_tf_example = tf.placeholder(dtype=tf.string,
                                         shape=[None],
                                         name='input_tensors')
  receiver_tensors = {'inputs': serialized_tf_example}
  features = tf.parse_example(serialized_tf_example, feature_spec)
  return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)

classifier.export_savedmodel('./savedNetworksEstimator', serving_input_receiver_fn)
```

Il faut noter que toutes les données sont sauvegardées dans un sous répertoire de celui que nous avons fourni. Le sous répertoire porte un nom correspondant à un timestamp obtenu lors de l'execution. notre programme complet renommera ce repertoire de sauvegarde en `./savedNetworksEstimator/lastSave`

Notons également la création d'un tenseur d'entrée pour les données qui porte le nom `input_tensors`.

Chargement

Pour le chargement, nous procédons en deux étapes, peu claires.

```
tf.saved_model.loader.load(sess, [tf.saved_model.tag_constants.SERVING], './
    savedNetworksEstimator/lastSave')

predictor= tf.contrib.predictor.from_saved_model('./savedNetworksEstimator/lastSave')
```

Enfin, nous récupérons le tenseur d'entrée de nos données :

```
input_tensor=tf.get_default_graph().get_tensor_by_name("input_tensors:0")
```

Pour introduire un exemple dans le réseau, il faut préparer les données

```
exemple = np.array([6.9, 3.2, 4.5, 1.5], dtype=np.float32)

model_input= tf.train.Example(features=tf.train.Features(feature={
                    'x': tf.train.Feature(float_list=tf.train.FloatList(value=exemple
    ))
                    }))

model_input=model_input.SerializeToString()
```

Enfin, on peu calculer les prédictions.

```
predictions= predictor({"inputs":[model_input]})
```

Cette prédiction est un dictionnaire dont la clef "scores" est ce qui nous intéresse :

```
classe_id = np.argmax(predictions["scores"])
```

A ce stade, nous ne savons pas récupérer les probabilités de chaque classe. Nous travaillons encore sur ce point, mais le classifieur est fonctionnel.

2.3.7 Programmes de sauvegarde et prédiction complets

Dans ces programmes, nous avons ajouté quelques lignes pour paramétrer proprement les noms des fichiers et répertoires de sauvegarde et une procédure qui efface les répertoires de sauvegarde des modèles s'ils existent.

Voici le code du programme qui réalise l'apprentissage, contenu dans le fichier : `DNN/Documentation/TutosPython/Iris/irisDnnEstimatorTrain.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen

import numpy as np
import shutil

import tensorflow as tf



# Data sets
IRIS_TRAINING = "IrisDatabase/iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "IrisDatabase/iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

# If the training and test sets aren't stored locally, download them.
if not os.path.exists(IRIS_TRAINING):
    raw = urlopen(IRIS_TRAINING_URL).read()
    with open(IRIS_TRAINING, "wb") as f:
        f.write(raw)

if not os.path.exists(IRIS_TEST):
    raw = urlopen(IRIS_TEST_URL).read()
    with open(IRIS_TEST, "wb") as f:
        f.write(raw)

# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TRAINING,
  target_dtype=np.int,
  features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename=IRIS_TEST,
  target_dtype=np.int,
  features_dtype=np.float32)

# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
```

```
# Build 3 layer DNN with 10, 20, 10 units respectively.

# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                        hidden_units=[10, 20, 10],
                                        n_classes=3,
                                        model_dir=visuPath)

# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(training_set.data)},
  y=np.array(training_set.target),
  num_epochs=None,
  shuffle=True)


# Train model.
classifier.train(input_fn=train_input_fn, steps=2000)


# Save Model

def serving_input_receiver_fn():
  feature_spec = {'x': tf.FixedLenFeature([4],tf.float32)}
  serialized_tf_example = tf.placeholder(dtype=tf.string,
                                         shape=[None],
                                         name='input_tensors')
  receiver_tensors = {'inputs': serialized_tf_example}
  features = tf.parse_example(serialized_tf_example, feature_spec)
  return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)

savePath = './SavedNetworksEstimator/'
if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

classifier.export_savedmodel(savePath, serving_input_receiver_fn)

## Supression du repertoire Timestamp, remplace par lastSave
folderName = os.listdir(savePath)[0]
folderFullName = os.path.join(savePath, folderName)
targetFullName = os.path.join(savePath, 'lastSave')

shutil.move(folderFullName,targetFullName)



# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(training_set.data)},
  y=np.array(training_set.target),
  num_epochs=1,
  shuffle=False)
```

```
accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("\nLearning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": np.array(test_set.data)},
  y=np.array(test_set.target),
  num_epochs=1,
  shuffle=False)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("\nTest Accuracy: {0:f}\n".format(accuracy_score))
```

../../TutosPython/Iris/irisDnnEstimatorTrain.py

Voici le code du programme qui réalise les prédictions, contenu dans le fichier : `DNN/Documentation/TutosPython/Iris/irisDnnEstimatorPredict.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import numpy as np

import tensorflow as tf


with tf.Session() as sess:

  # Chargement du réseau
  basePath = 'SavedNetworksEstimator'
  tmpDir = 'lastSave'
  savePathFull = os.path.join(basePath, tmpDir)
  print ("Restoring from ", savePathFull)

  # loading model
  tf.saved_model.loader.load(sess, [tf.saved_model.tag_constants.SERVING],
    savePathFull)

  #get the predictor , refer tf.contrib.predicdtor
  predictor= tf.contrib.predictor.from_saved_model(savePathFull)

  #get the input_tensor tensor from the model graph
  # name is input_tensor defined in input_receiver function refer to tf.dnn.
    classifier
  input_tensor=tf.get_default_graph().get_tensor_by_name("input_tensors:0")

  #get the output dict
  # do not forget [] around model_input or else it will complain shape() for Tensor
    shape(?,)
  # since its of shape(?,) when we trained it

  new_samples = np.array(
  [[6.9, 3.2, 4.5, 1.5],
   [4.8, 3.1, 5.0, 1.7]], dtype=np.float32)

  for sample in new_samples:
```

```
        print (sample)

        model_input= tf.train.Example(features=tf.train.Features(feature={
                        'x': tf.train.Feature(float_list=tf.train.FloatList(value=sample)
    )
                        }))

        #Prepare model input, the model expects a float array to be passed to x
        # check line 28 serving_input_receiver_fn
        model_input=model_input.SerializeToString()

        # calcul de la prediction ... depuis les scores
        predictions= predictor({"inputs":[model_input]})
        classe_id = np.argmax(predictions["scores"])

        dicoClasses = ['setosa', 'versicolor', 'virginica']

        print ("je pense que c'est : ",dicoClasses[classe_id])
```

../../TutosPython/Iris/irisDnnEstimatorPredict.py

2.4 Performances

Pour information, voici des performances standard obtenues sur la base Iris. Monocouche :

```
Resultats en Apprentissage 0.98333335
Résultats en Généralisation 0.96666664
```

Multicouches :

```
Resultats en Apprentissage 0.991667
Résultats en Généralisation 0.966667
```

Chapitre 3

bases MNIST et Fashion MNIST

3.1 La Base MNIST

La base MNIST est bien connue, il s'agit de reconnaitre des chiffres (10 classes), a partir d'images de 28x28 pixels, que dans un premier temps nous transformerons en un vecteur de 784 valeurs.

3.1.1 Lecture des données

Lecture des données : Nous importons un module permettant de récupérer la base MNIST, puis on demande à ce module de lire les données. Si elles ne sont pas présentes, il va les télécharger sur le site de MNIST (`http://yann.lecun.com/exdb/mnist/`)

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/")
```

L'objet contenant nos données permet l'accès aux données d'apprentissage dans `mnist.train.images` et `mnist.train.labels`. Les données de généralisations sont accessible via `mnist.test.images` et `mnist.test.labels`.

Lorsque nous voudrons faire de la prédiction, nous partirons d'une image quelconque, qu'il faudra préparer (transformer en image 28x28, inverser le contraste...) On pourra le faire comme suit :

```
imageFilename = 'images/bidon.jpg'
imageGray = Image.open(imageFilename).resize((28,28)).convert('L')
imageInvert =  PIL.ImageOps.invert(imageGray)

#imageInvert.save('temp.bmp')


# conversion en vecteur
a = np.array(imageInvert)
flat_arr = a.reshape((1, 784))
```

Ce dernier vecteur pourra être introduit dans les réseaux.

3.1.2 Mnist : Réseau monocouche boite blanche

Construction du modèle

On utilisera ici un modèle purement linéaire contenant une couche de 10 neurones (c'est la couche de sortie) tout à fait semblable à celui de la base Iris vu dans la section 2.2.2. Simplement,

il y a maintenant 10 neurones et 784 entrées.

```
# entrées
x = tf.placeholder(tf.float32, [None, 784])

# sorties voulues
y_ = tf.placeholder(tf.float32, [None, 10])

# Le modèle
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# sorties calcules
score = tf.matmul(x, W) + b
```

Les calculs d'entropie croisée et de précision sont ceux vu dans la section 2.2.2

Lancement des calculs

L'apprentissage est très semblable à celui que nous avons fait pour la base Iris, a ceci prêt que nous tirons profit de l'objet mnist qui permet de faire des batch de 100 exemples.

Ici, nous proposons un apprentissage en 1000 passes sur 100 exemples tirés au sort.

```
# Apprentissage
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

L'évaluation est classique. Par exemple, en généralisation :

```
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: mnist.test.
    images,     y_: mnist.test.labels}))
```

Pour la prédiction, il faudra lire une image, la transformer en vecteur de taille 784 comme on l'a vu dans la section 3.1.1 Ce dernier vecteur est alors introduit dans le réseau pour prédiction

```
print("\nJe pense que c'est un ", sess.run(classe, {x: flat_arr}))
```

Codes complets

Le code complet, incluant des instructions de mise en forme pour TensorBoard est le suivant. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistMonocoucheComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from PIL import Image
import PIL.ImageOps


import numpy as np
```

```
import tensorflow as tf

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 784], name = "X")

with tf.name_scope('Y_True'):
    # sorties voulues
    y_ = tf.placeholder(tf.float32, [None, 10], name = "Y_True")

# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([784, 10]),name ="W")

with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([10]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b

classe = tf.argmax(score,1, name="classe")

# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
    y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#   cross_entropy = tf.reduce_mean(
#       tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)


sess = tf.Session()


# Configuration de TensorBoard
# If the visu dir exists, we delete it.
# to avoid accidental multiple trainings visualisation
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)
```

```
writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()



init = tf.global_variables_initializer();
sess.run(init);

for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  summary, _ = sess.run([merged,train_step], feed_dict={x: batch_xs, y_: batch_ys})
  #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
     batch_ys}))

  writer.add_summary(summary, i)



print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: mnist.train.
    images, y_: mnist.train.labels}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: mnist.test.
    images, y_: mnist.test.labels}))

## Prediction sur une image
# Lecture de l'image, et préparation de l'image
imageFilename = 'images/5.png'
imageGray = Image.open(imageFilename).resize((28,28)).convert('L')
imageInvert =  PIL.ImageOps.invert(imageGray)

#imageInvert.save('temp.bmp')


# conversion en vecteur
a = np.array(imageInvert)
flat_arr = a.reshape((1, 784))

print("\nJe pense que c'est un ", sess.run(classe, {x: flat_arr}))

writer.close()
```

../../TutosPython/Mnist/mnistMonocoucheComplet.py

Ci dessous, le code complet du programme en charge de l'apprentissage et de la sauvegarde du modèle. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistMonocoucheTrain.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import shutil

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
import tensorflow as tf

with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 784], name = "X")

with tf.name_scope('Y_True'):
    # sorties voulues
    y_ = tf.placeholder(tf.float32, [None, 10], name = "Y_True")

# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([784, 10]),name ="W")

with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([10]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b

with tf.name_scope("Classe"):
    classe = tf.argmax(score,1, name="classe")

# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
    softmax = y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#    cross_entropy = tf.reduce_mean(
#        tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name="accuracy
    ")

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.Session()


# Configuration de TensorBoard
# If the visu dir exists, we delete it.
# to avoid accidental multiple trainings visualisation
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)
```

```
#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()



init = tf.global_variables_initializer();
sess.run(init);

for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  summary, _ = sess.run([merged,train_step], feed_dict={x: batch_xs, y_: batch_ys})
  #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
     batch_ys}))

  writer.add_summary(summary, i)

#Create a saver object which will save all the variables
saver = tf.train.Saver()

# sauvegarde en fin d'apprentissage
savePath = 'SavedNetworks/'
modelName = 'myMonoCouchemodel.ckpt'

if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

savePathFull = os.path.join(savePath, modelName)

saver.save(sess, savePathFull)


print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: mnist.train.
    images, y_: mnist.train.labels}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: mnist.test.
    images, y_: mnist.test.labels}))

writer.close()
```

../../TutosPython/Mnist/mnistMonocoucheTrain.py

Enfin, voici le code complet du programme en charge du chargement d'un réseau et de la réalisation des prédictions. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistMonocouchePredict.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import shutil

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf

with tf.name_scope('X'):
    # entrées
    x = tf.placeholder(tf.float32, [None, 784], name = "X")
```

```
with tf.name_scope('Y_True'):
    # sorties voulues
    y_ = tf.placeholder(tf.float32, [None, 10], name = "Y_True")

# Le modèle
with tf.name_scope("Weights"):
    W = tf.Variable(tf.zeros([784, 10]),name ="W")

with tf.name_scope("Biases"):
    b = tf.Variable(tf.zeros([10]), name = "b")

with tf.name_scope("Score"):
    score = tf.matmul(x, W) + b

with tf.name_scope("Classe"):
    classe = tf.argmax(score,1, name="classe")

# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
    softmax = y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#   cross_entropy = tf.reduce_mean(
#       tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name="accuracy
    ")

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.Session()


# Configuration de TensorBoard
# If the visu dir exists, we delete it.
# to avoid accidental multiple trainings visualisation
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()
```

```
init = tf.global_variables_initializer();
sess.run(init);

for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  summary, _ = sess.run([merged,train_step], feed_dict={x: batch_xs, y_: batch_ys})
  #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
     batch_ys}))

  writer.add_summary(summary, i)

#Create a saver object which will save all the variables
saver = tf.train.Saver()

# sauvegarde en fin d'apprentissage
savePath = 'SavedNetworks/'
modelName = 'myMonoCouchemodel.ckpt'

if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

savePathFull = os.path.join(savePath, modelName)

saver.save(sess, savePathFull)


print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: mnist.train.
    images, y_: mnist.train.labels}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: mnist.test.
    images, y_: mnist.test.labels}))

writer.close()
```

../../TutosPython/Mnist/mnistMonocoucheTrain.py

3.1.3 Mnist : Réseau multicouches boite noire

Le lecteur peut se référer à la section 2.3.2 pour la construction d'un réseau multicouche avec Estimator. Les principes ayant été vu, nous nous bornerons a donner le code complet des trois programmes : Complet, Train, Predict.

Codes complets

Le code complet, incluant des instructions de mise en forme pour TensorBoard est le suivant. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistDnnEstimatorComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen
import shutil

from PIL import Image
import PIL.ImageOps
```

```
import numpy as np
import tensorflow as tf

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data/')


# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[784])]

# sauvegarde en fin d'apprentissage
# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.estimator.DNNClassifier(
 feature_columns=feature_columns,
 hidden_units=[256, 32],
 optimizer=tf.train.AdamOptimizer(1e-4),
 n_classes=10,
 dropout=0.1,
 model_dir=visuPath
)

def input(dataset):
 return dataset.images, dataset.labels.astype(np.int32)

def getFeatures(dataset):
 return dataset.images

def getLabels(dataset):
 return dataset.labels.astype(np.int32)


# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=None,
 batch_size=50,
 shuffle=True
)

# Train model.
classifier.train(input_fn=train_input_fn, steps=10000)


# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=1,
 shuffle=False
)
```

```
accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("Learning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.test)},
 y=getLabels(mnist.test),
 num_epochs=1,
 shuffle=False
)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("Test Accuracy: {0:f}\n".format(accuracy_score))


## Prediction sur une image
# Lecture de l'image, et préparation de l'image
imageFilename = 'images/flou.jpg'
imageGray = Image.open(imageFilename).resize((28,28)).convert('L')
imageInvert =  PIL.ImageOps.invert(imageGray)

#imageInvert.save('temp.bmp')


# conversion en vecteur
a = np.array(imageInvert)
flat_arr = a.reshape((1, 784))

predict_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": flat_arr},
  num_epochs=1,
  shuffle=False)

predictions = classifier.predict(input_fn=predict_input_fn)

for p in predictions :
    class_id = p['class_ids'][0]
    probability = p['probabilities'][class_id]
    print ("je pense que c'est un : ",class_id, "avec une proba de ",probability )
```

../../TutosPython/Mnist/mnistDnnEstimatorComplet.py

Ci dessous, le code complet du programme en charge de l'apprentissage et de la sauvegarde du modèle. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistDnnEstimatorTrain.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen
import shutil

from PIL import Image
import PIL.ImageOps

import numpy as np

import tensorflow as tf
```

```
# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data/')


# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[784])]

# sauvegarde en fin d'apprentissage
# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.estimator.DNNClassifier(
 feature_columns=feature_columns,
 hidden_units=[256, 32],
 optimizer=tf.train.AdamOptimizer(1e-4),
 n_classes=10,
 dropout=0.1,
 model_dir=visuPath
)

def input(dataset):
 return dataset.images, dataset.labels.astype(np.int32)

def getFeatures(dataset):
 return dataset.images

def getLabels(dataset):
 return dataset.labels.astype(np.int32)


# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=None,
 batch_size=50,
 shuffle=True
)

# Train model.
classifier.train(input_fn=train_input_fn, steps=10000)


# Save Model

def serving_input_receiver_fn():
  feature_spec = {'x': tf.FixedLenFeature([784],tf.float32)}
  serialized_tf_example = tf.placeholder(dtype=tf.string,
                                         shape=[None],
                                         name='input_tensors')
  receiver_tensors = {'inputs': serialized_tf_example}
  features = tf.parse_example(serialized_tf_example, feature_spec)
  return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)
```

```
savePath = './SavedNetworksEstimator'
if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

classifier.export_savedmodel(savePath, serving_input_receiver_fn)

## Supression du repertoire Timestamp, remplace par lastSave
folderName = os.listdir(savePath)[0]
folderFullName = os.path.join(savePath, folderName)
targetFullName = os.path.join(savePath, 'lastSave')

shutil.move(folderFullName,targetFullName)


# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=1,
 shuffle=False
)

accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("Learning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.test)},
 y=getLabels(mnist.test),
 num_epochs=1,
 shuffle=False
)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("Test Accuracy: {0:f}\n".format(accuracy_score))
```

../../TutosPython/Mnist/mnistDnnEstimatorTrain.py

Enfin, voici le code complet du programme en charge du chargement d'un réseau et de la réalisation des prédictions. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/Mnist/mnistDnnEstimatorPredict.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen
import shutil

from PIL import Image
import PIL.ImageOps

import numpy as np

import tensorflow as tf

# récupération des bases de données
```

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data/')


# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[784])]

# sauvegarde en fin d'apprentissage
# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.estimator.DNNClassifier(
 feature_columns=feature_columns,
 hidden_units=[256, 32],
 optimizer=tf.train.AdamOptimizer(1e-4),
 n_classes=10,
 dropout=0.1,
 model_dir=visuPath
)

def input(dataset):
 return dataset.images, dataset.labels.astype(np.int32)

def getFeatures(dataset):
 return dataset.images

def getLabels(dataset):
 return dataset.labels.astype(np.int32)


# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=None,
 batch_size=50,
 shuffle=True
)

# Train model.
classifier.train(input_fn=train_input_fn, steps=10000)


# Save Model

def serving_input_receiver_fn():
  feature_spec = {'x': tf.FixedLenFeature([784],tf.float32)}
  serialized_tf_example = tf.placeholder(dtype=tf.string,
                                         shape=[None],
                                         name='input_tensors')
  receiver_tensors = {'inputs': serialized_tf_example}
  features = tf.parse_example(serialized_tf_example, feature_spec)
  return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)

savePath = './SavedNetworksEstimator'
```

```
if os.path.exists(savePath):
  shutil.rmtree(savePath)
os.makedirs(savePath)

classifier.export_savedmodel(savePath, serving_input_receiver_fn)

## Supression du repertoire Timestamp, remplace par lastSave
folderName = os.listdir(savePath)[0]
folderFullName = os.path.join(savePath, folderName)
targetFullName = os.path.join(savePath, 'lastSave')

shutil.move(folderFullName,targetFullName)


# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=1,
 shuffle=False
)

accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("Learning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.test)},
 y=getLabels(mnist.test),
 num_epochs=1,
 shuffle=False
)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("Test Accuracy: {0:f}\n".format(accuracy_score))
```

../../TutosPython/Mnist/mnistDnnEstimatorTrain.py

3.1.4 Performances

Pour information, voici des performances standard obtenues sur la base MNIST. Monocouche :

```
Résultats en Apprentissage 0.91827273
Résultats en Généralisation 0.9178
```

Multicouches :

```
Resultats en Apprentissage 0.98333335
Résultats en Généralisation 0.96666664
```

3.2 La base Fashion MNIST

La base Fashion MNIST ressemble beaucoup à la base MNIST :

— Même nombre d'entrées
— Même nombre de classes
— Même nombre d'exemples
— Même noms de fichiers

3.2.1 Lecture des données

De ce fait, nous pouvons utiliser les mêmes procédures pour charger les exemples que celle utilisées pour MNIST. Ces procédures font partie du module exemple de tensorflow que l'on charge comme suit :

```
from tensorflow.examples.tutorials.mnist import input_data
```

La seule modification effective consiste a spécifier le répertoire ou l'on trouve les fichiers de Fashion MNIST :

```
# Import data
fashionMnist = input_data.read_data_sets('./FM_DATA/')
```

C'est cet objet `fashionMnist` qui fournira les données et les labels lors de l'entrainement et de l'évaluation.

ATTENTION : Nous utilisons içi un module conçu pour MNIST pour lire les données de Fashion MNIST. Conformément à ce qui a été dit dans la section 3.1.1, si le répertoire `./FM_DATA/` est vide ou n'existe pas, ce module va télécharger la base MNIST dedans ! Le téléchargement de Fashion MNIST doit donc être fait manuellement, depuis
`https://github.com/zalandoresearch/fashion-mnist/`.

3.2.2 Fmnist : Réseau monocouche boite blanche

Compte tenu de ce qui a été dit plus haut, le code est quasiment le même que dans le cas de MNIST vu dans la section **??**. Voici donc le code complet du programme. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/FashionMNIST/fmnistMonocoucheComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from PIL import Image
import PIL.ImageOps
import numpy as np
import os
import shutil

import tensorflow as tf

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data
fashionMnist = input_data.read_data_sets('./FM_DATA/', one_hot=True)


with tf.name_scope('X'):
  # entrées
```

```
  x = tf.placeholder(tf.float32, [None, 784], name = "X")

with tf.name_scope('Y_True'):
  # sorties voulues
  y_ = tf.placeholder(tf.float32, [None, 10], name = "Y_True")

# Le modèle
with tf.name_scope("Weights"):
  W = tf.Variable(tf.zeros([784, 10]),name ="W")

with tf.name_scope("Biases"):
  b = tf.Variable(tf.zeros([10]), name = "b")

with tf.name_scope("Score"):
  score = tf.matmul(x, W) + b

classe = tf.argmax(score,1)


# calcul de l'entropie croisée
# Cross entropy version 1 (un peu instable)
with tf.name_scope('softmax'):
  softmax = y = tf.nn.softmax(score)
with tf.name_scope('cross_entropy'):
  cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices
    =[1]))

# Cross entropy version 2
#with tf.name_scope('cross_entropy'):
#  cross_entropy = tf.reduce_mean(
#      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=score))

# Calcul de la prédiction finale
with tf.name_scope('Accuracy'):
  correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Choix d'une méthode de minimisation
with tf.name_scope('train'):
  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

sess = tf.Session()


# Configuration de TensorBoard
# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuMonoCouche'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

writer = tf.summary.FileWriter(visuPath, sess.graph)
tf.summary.scalar('Entropie Croisee', cross_entropy)
tf.summary.scalar('Precision', accuracy)

#tf.summary.scalar('W', W)
merged = tf.summary.merge_all()
```

```
init = tf.global_variables_initializer();
sess.run(init);

for i in range(1000):
  batch_xs, batch_ys = fashionMnist.train.next_batch(1000)
  summary, _ = sess.run([merged,train_step], feed_dict={x: batch_xs, y_: batch_ys})
  #print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: batch_xs, y_:
     batch_ys}))

  writer.add_summary(summary, i)



print("Resultats en Apprentissage", sess.run(accuracy, feed_dict={x: fashionMnist.
    train.images, y_: fashionMnist.train.labels}))
print("Résultats en Généralisation", sess.run(accuracy, feed_dict={x: fashionMnist.
    test.images, y_: fashionMnist.test.labels}))

## Prediction sur une image
# Lecture de l'image, et préparation de l'image
imageFilename = 'images/tshirt.jpg'
imageGray = Image.open(imageFilename).resize((28,28)).convert('L')
imageInvert =  PIL.ImageOps.invert(imageGray)

#imageInvert.save('temp.bmp')


# conversion en vecteur
a = np.array(imageInvert)
flat_arr = a.reshape((1, 784))

dicoClasses = ["t-shirts", "trousers", "pullovers", "dresses", "coats", "sandals", "
    shirts", "sneakers", "bags", "ankle boots"]

classIndex = sess.run(classe, {x: flat_arr})
print("\nJe pense que c'est : ", dicoClasses[classIndex[0]], " / label : ",
    classIndex)


writer.close()
```

../../TutosPython/FashionMNIST/fmnistMonocoucheComplet.py

3.2.3 Fmnist : Réseau multicouche boite noire

Compte tenu de ce qui a été dit plus haut, le code est quasiment le même que dans le cas de MNIST vu dans la section 3.1.3. Voici donc le code complet du programme. Ce code est contenu dans le fichier :
`DNN/Documentation/TutosPython/FashionMNIST/fmnistDnnEstimatorComplet.py`

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from six.moves.urllib.request import urlopen

from PIL import Image
import PIL.ImageOps
```

```
import numpy as np
import shutil

import tensorflow as tf

# récupération des bases de données
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./FM_DATA/')


# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[784])]

# Build 3 layer DNN with 10, 20, 10 units respectively.
# If the model_dir exists, we delete it.
# to avoid accidental multiple trainings.
visuPath = './VisuDnn'
if os.path.exists(visuPath):
  shutil.rmtree(visuPath)
os.makedirs(visuPath)

classifier = tf.estimator.DNNClassifier(
 feature_columns=feature_columns,
 hidden_units=[256, 32],
 optimizer=tf.train.AdamOptimizer(1e-4),
 n_classes=10,
 dropout=0.1,
 model_dir=visuPath
)

def input(dataset):
 return dataset.images, dataset.labels.astype(np.int32)

def getFeatures(dataset):
 return dataset.images

def getLabels(dataset):
 return dataset.labels.astype(np.int32)


# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=None,
 batch_size=50,
 shuffle=True
)

# Train model.
classifier.train(input_fn=train_input_fn, steps=10000)

# Evaluation sur la base d'apprentissage
train_input_eval_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.train)},
 y=getLabels(mnist.train),
 num_epochs=1,
 shuffle=False
)
```

```
accuracy_score = classifier.evaluate(input_fn=train_input_eval_fn)["accuracy"]
print("\nLearning Accuracy: {0:f}\n".format(accuracy_score))

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
 x={"x": getFeatures(mnist.test)},
 y=getLabels(mnist.test),
 num_epochs=1,
 shuffle=False
)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("\nTest Accuracy: {0:f}\n".format(accuracy_score))

## Prediction sur une image

# Lecture de l'image, et préparation de l'image
imageFilename = 'images/nastase.jpg'
imageGray = Image.open(imageFilename).resize((28,28)).convert('L')
imageInvert =  PIL.ImageOps.invert(imageGray)

#imageInvert.save('temp.bmp')


# conversion en vecteur
a = np.array(imageInvert)
flat_arr = a.reshape((1, 784))

predict_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": flat_arr},
  num_epochs=1,
  shuffle=False)

predictions = classifier.predict(input_fn=predict_input_fn)

dicoClassesEn = ["un t-shirt", "un pantalon trousers", "pullovers", "dresses", "coats
    ", "sandals", "shirts", "sneakers", "bags", "ankle boots"]
dicoClasses = ["un t-shirt", "un pantalon", "un pull", "une robe", "un manteau", "une
     sandale", "une chemise", "une basket", "un sac", "une botte"]


for p in predictions :
    class_id = p['class_ids'][0]
    probability = p['probabilities'][class_id]
    print ("je pense que c'est : ",dicoClasses[class_id], "avec une proba de ",
    probability )
```

../../TutosPython/FashionMNIST/fmnistDnnEstimatorComplet.py

3.2.4 Performances

Pour information, voici des performances standard obtenues sur la base Fashion MNIST. Monocouche :

```
Resultats en Apprentissage 0.79394543
Résultats en Généralisation 0.7783
```

Multicouches :

```
Resultats en Apprentissage 0.889000
Résultats en Généralisation 0.868800
```

On peut noter la baisse de performances par rapport à ce qui était obtenu avec la base MNIST dans la section 3.1.4.