



# Tutoriel TensorFlow

## Prise en main et installation de la bibliothèqueTensorFlow.

M. IKNI Layachi

Janvier 2018 à Février 2018

L3 Informatique

**Tuteur :** M.PAGE Vincent **.**

**Enseignant Référent :** Mm. Suzy GAUCHER-CAZALIS

## Etablissement :Université des Antilles .

## Etablissement d'accueil: Laboratoire LAMIA .

## Installation

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

Pour l'installation, nous avons suivi les instructions du tutoriel officiel qui se trouve ici, sans difficultés : https://www.tensorflow.org/install/

## Premiers concepts de TensorFlow

Tout d'abord, TensorFlow s'appuie sur des concepts de programmation très différents d'une programmation standard python. Pour bien les comprendre, prenons un exemple :

On veut que notre programme prend une valeur réelle (x), calcule une valeur y = W\*x+b avec W et b des valeurs réelles que notre programme sera appelé a 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 a 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**)**

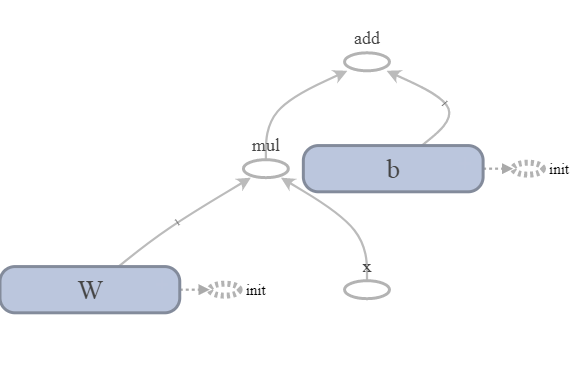
La sortie de ce programme est alors surprenante :

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.



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 (premiere 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 :

**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**)**

## Premiers pas avec Tensorboard

TensorBoard est l'outil de visualisation associé a 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**=**"./pathLog/"**;**

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.

On lance notre programme TensorFlow

python .\premiersPas.py

On lance ensuite TensorBoard sur le **répertoire de Log**

tensorboard.exe --logdir=./pathLog

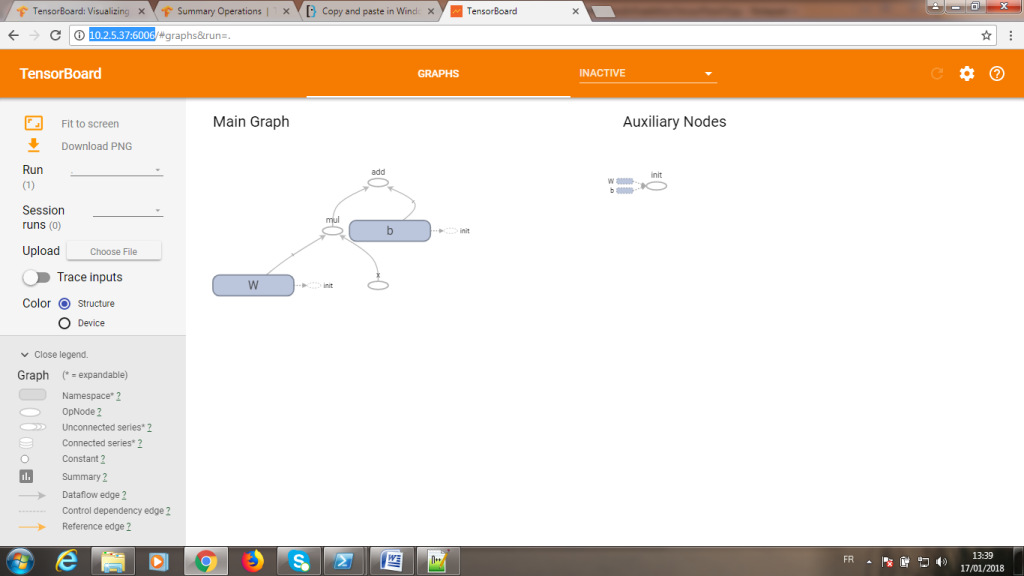
Enfin, on ouvre un navigateur dans lequel on indique l'URL de consultation. Cette url correspond a l'adresse IP de la machine sur laquelle on travaille, et TensorBoard utilise le port 6006 pour son serveur web.

Cette adresse IP est obtenue sous windows avec la commande

ipconfig

Notre IP étant 10.2.5.37, l'url complète est http://10.2.5.37:6006

Voici une capture de la visualisation obtenue :



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.

## Premier apprentissage automatique

Il s'agit du problème classique de **régression linéaire**. on se donne un ensemble de valeurs de x, et un ensemble de valeurs attendues pour chacun de ces x. L'objectif est de trouver la droite qui passe le plus près de tous ces points.

**import** tensorflow **as** tf

x **=** tf**.**placeholder**(**tf**.**float32**)**

sortieVoulue **=** tf**.**placeholder**(**tf**.**float32**)**

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

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

sortieCalculee **=** W**\***x **+** b

Nous avons bien défini nos entrées :

* x : un placeholder dans lequel nous placerons au moment du run les abscisses des points
* y : un placeholder dans lequel nous placerons au moment du run les ordonnées des points

Par ailleurs, nous allons chercher des droites, qui sont paramétrées par leur pente (W) et leur abscisse à l'origine (b). Nous avons fixé les valeurs initiales de W et b a respectivement 0.3 et -0.3. Pour un couple (W,b), nous avons une droite qui à x associe sortieCalculee.

Notre problème consiste donc a modifier W et b pour que sortieCalculee soit aussi prêt que possible de sortieVoulue.

Pour un couple (W,b), évaluons l'erreur que l'on commet. on peut mesurer la somme des erreurs quadratiques sur chaque entrée :

squared\_deltas **=** tf**.**square**(**sortieCalculee **-** sortieVoulue**)**

erreur **=** tf**.**reduce\_sum**(**squared\_deltas**)**

Ajoutons enfin le run qui permettra de faire ces calculs pour les valeurs initiales de W et b :

sess **=** tf**.**Session**()**

init **=** tf**.**global\_variables\_initializer**()**

sess**.**run**(**init**)**

**print("erreur :",**sess**.**run**(**erreur**,{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]}))**

Pour nos valeurs initiales, l'erreur totale est donnée par la sortie suivante

erreur : 23.66

De fait, quelques calculs rapides permettrait de trouver les meilleures valeurs possible de W,b : (-1, 1). Dans le cas particulier des x et sortieVoulue donnés, il existe réellement une droite qui passe par ces points...

Essayons de vérifier cela

fixW **=** tf**.**assign**(**W**,** **[-**1.**])**

fixb **=** tf**.**assign**(**b**,** **[**1.**])**

sess**.**run**([**fixW**,** fixb**])**

**print(**"erreur :"**,**sess**.**run**(**erreur**,{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]}))**

Les deux premières lignes créent chacune un noeud dont l'objectif est de corriger les valeurs de W et b. La troisième ligne lance le calcul sur ces noeuds, modifiant ainsi les valeurs de nos variables

Enfin la derniere ligne calcule l'erreur pour les x et sortieVoulue fournis avec le résultat suivant :

erreur : 0.0

Essayons maintenant de trouver automatiquement ces valeurs pour W et b :

optimizer **=** tf**.**train**.**GradientDescentOptimizer**(**0.01**)**

train **=** optimizer**.**minimize**(**erreur**)**

sess**.**run**(**init**)** # reset values to incorrect defaults.

**for** i **in** range**(**1000**):**

sess**.**run**(**train**,** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]})**

**print("(W,b finaux) :",** sess**.**run**([**W**,** b**]))**

Pour cela, nous utilisons un objet optimizer prédéfini par TensorFlow, et qui opèrera une descente de gradient (avec un pas de 0.0.1). on définit un nœud de calcul (train) dont le but est de minimiser l'erreur .

la ligne suivante réinitialise les valeurs de W et b a (0.3, -0.3)

puis, 1000 fois de suite, on calcule le nœud train. Chaque itération améliore un peu les valeurs (W et b). Il faut noter que nulle part, on ne spécifie que les valeurs à modifier sont W et b... le programme le devine à partir du graphe : W et b sont les seules variables du graphe....

Enfin la dernière ligne affiche les valeurs finales trouvées :

(W,b finaux) : [array([-0.9999969], dtype=float32), array([0.9999908], dtype=float32)]

On voit que notre programme a trouvé pour W et b des valeurs très proches des valeurs cherchées (-1,1)

Voici donc le code complet de ce programme

**import** tensorflow **as** tf

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

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

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

sortieCalculee **=** W**\***x **+** b

sortieVoulue **=** tf**.**placeholder**(**tf**.**float32**,** name**=**"sortieVoulue"**)**

squared\_deltas **=** tf**.**square**(**sortieCalculee **-** sortieVoulue**)**

erreur **=** tf**.**reduce\_sum**(**squared\_deltas**)**

sess **=** tf**.**Session**()**

init **=** tf**.**global\_variables\_initializer**()**

sess**.**run**(**init**)**

**print(**sess**.**run**(**sortieCalculee**,** **{**x**:** **[**1**,** 2**,** 3**,** 4**]}))**

**print(**"erreur totale :"**,** sess**.**run**(**erreur**,** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]}))**

fixW **=** tf**.**assign**(**W**,** **[-**1.**])**

fixb **=** tf**.**assign**(**b**,** **[**1.**])**

sess**.**run**([**fixW**,** fixb**])**

**print(**"erreur totale :"**,** sess**.**run**(**erreur**,** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]}))**

optimizer **=** tf**.**train**.**GradientDescentOptimizer**(**0.01**)**

train **=** optimizer**.**minimize**(**erreur**)**

sess**.**run**(**init**)** # reset values to incorrect defaults.

**for** i **in** range**(**1000**):**

sess**.**run**(**train**,** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]})**

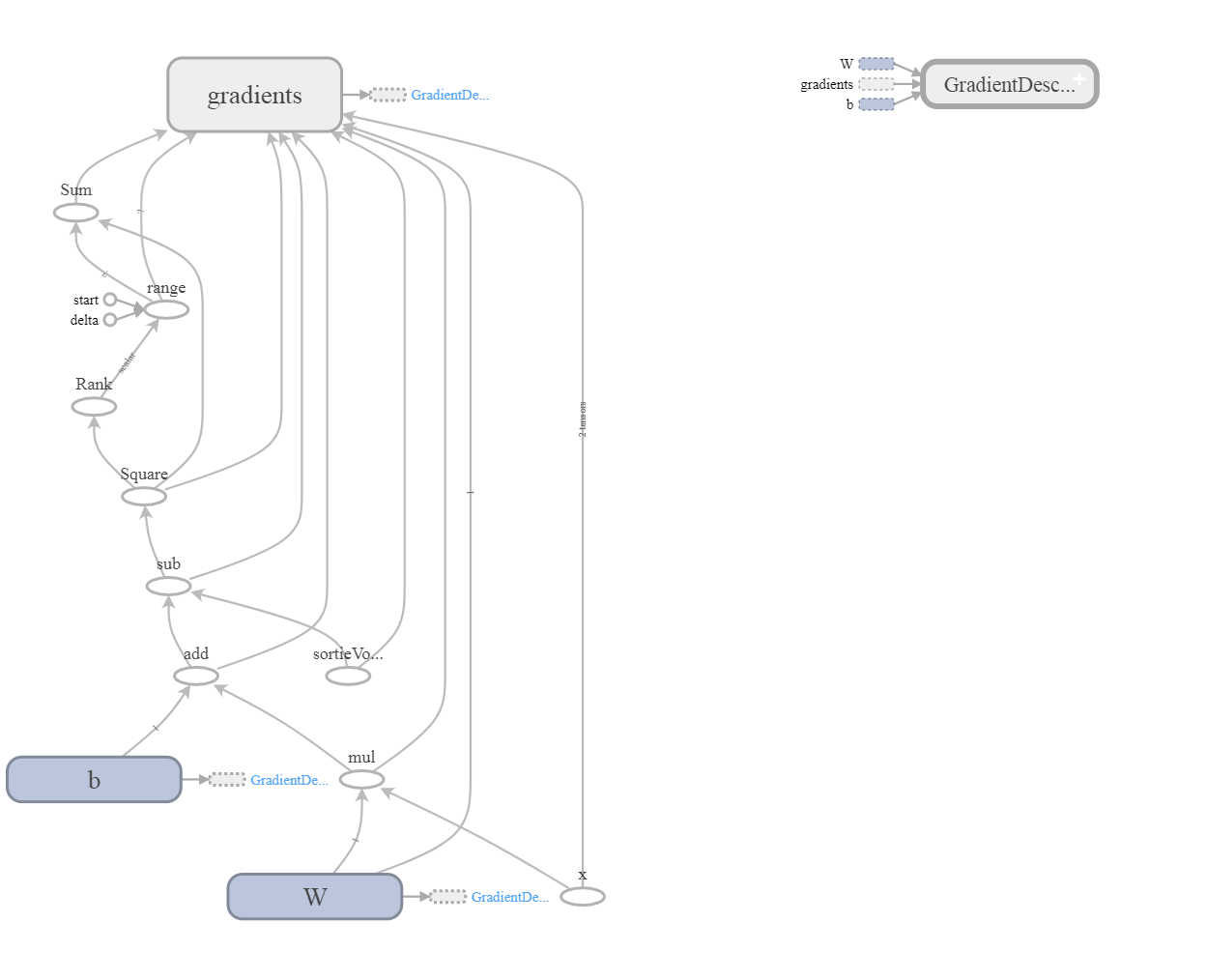
**print(**"(W,b finaux) :"**,** sess**.**run**([**W**,** b**]))**

pathLog**=**"./pathLog/"**;**

writer **=** tf**.**summary**.**FileWriter**(**pathLog**,** sess**.**graph**)**

writer**.**close**()**

Voici également le graphe de calcul que notre programme a utilisé



En l'état, une partie de ce graphe est facile a expliquer (en bas a gauche), le reste est assez... mysterieux mais est clairement associé a la descente du gradient...

## Suivi de performance dans TensorBoard

Nous voulons suivre l'evolution de l'erreur au cours de l'apprentissage des W,b du programme précédent et la visualiser a l'aide de TensorBoard.

Il va falloir modifier notre programme TensorFlow pour logger cette erreur. Les lignes à insérer étant réparties un peu partout, nous indiquerons chaque modification séparément et fourniront le code complet en fin de section.

tout d'abord, que l'on s'intéresse à l'erreur. Ceci est fait par la ligne suivante :

tf**.**summary**.**scalar**(**'erreur quadratique'**,** erreur**)**

De ce que nous avons compris, on peut s'interesser a plusieurs paramètres, et il faudra les fusionner pour les logger avec la ligne suivante :

merged **=** tf**.**summary**.**merge\_all**()**

Lors de chaque run, pendant l'entrainement, il faudra demander le calcul de ce merged. Enfin, le résumé devra etre sauvé grace au writer de TensorBoard...

**for** i **in** range**(**1000**):**

summary**,** \_ **=** sess**.**run**([**merged**,**train**],** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]})**

writer**.**add\_summary**(**summary**,** i**)**

Si une partie des notions utilisées ici nous échappe, voici néanmoins un code complet, fonctionnel :

**import** tensorflow **as** tf

## Définition des variables

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

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

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

sortieCalculee **=** W**\***x **+** b

sortieVoulue **=** tf**.**placeholder**(**tf**.**float32**,** name**=**"sortieVoulue"**)**

squared\_deltas **=** tf**.**square**(**sortieCalculee **-** sortieVoulue**)**

erreur **=** tf**.**reduce\_sum**(**squared\_deltas**)**

optimizer **=** tf**.**train**.**GradientDescentOptimizer**(**0.01**)**

train **=** optimizer**.**minimize**(**erreur**)**

sess **=** tf**.**Session**()**

# Configuration de TensorBoard

pathLog**=**"./pathLog/"**;**

writer **=** tf**.**summary**.**FileWriter**(**pathLog**,** sess**.**graph**)**

tf**.**summary**.**scalar**(**'erreur quadratique'**,** erreur**)**

#tf.summary.scalar('W', W)

merged **=** tf**.**summary**.**merge\_all**()**

init **=** tf**.**global\_variables\_initializer**()**

sess**.**run**(**init**)** # reset values to incorrect defaults.

**for** i **in** range**(**1000**):**

summary**,** \_ **=** sess**.**run**([**merged**,**train**],** **{**x**:** **[**1**,** 2**,** 3**,** 4**],** sortieVoulue**:** **[**0**,** **-**1**,** **-**2**,** **-**3**]})**

writer**.**add\_summary**(**summary**,** i**)**

**print(**"(W,b finaux) :"**,** sess**.**run**([**W**,** b**]))**

writer**.**close**()**

Voici le résultat final



## Première utilisation de TfEstimator

tf.estimator est une bibliothèque TensorFlow de haut niveau qui simplifie la mécanique de l'apprentissage automatique, notamment:

* exécuter des boucles d'entraînement
* exécuter des boucles d'évaluation
* gérer des ensembles de données

tf.estimator définit de nombreux modèles courants.

Utilisons le ici pour le problème précédent de régression Lineaire

# NumPy is often used to load, manipulate and preprocess data.

**import** numpy **as** np

**import** tensorflow **as** tf

# Declare list of features. We only have one numeric feature. There are many

# other types of columns that are more complicated and useful.

feature\_columns **=** **[**tf**.**feature\_column**.**numeric\_column**(**"x"**,** shape**=[**1**])]**

# An estimator is the front end to invoke training (fitting) and evaluation

# (inference). There are many predefined types like linear regression,

# linear classification, and many neural network classifiers and regressors.

# The following code provides an estimator that does linear regression.

estimator **=** tf**.**estimator**.**LinearRegressor**(**feature\_columns**=**feature\_columns**)**

# TensorFlow provides many helper methods to read and set up data sets.

# Here we use two data sets: one for training and one for evaluation

# We have to tell the function how many batches

# of data (num\_epochs) we want and how big each batch should be.

x\_train **=** np**.**array**([**1.**,** 2.**,** 3.**,** 4.**])**

y\_train **=** np**.**array**([**0.**,** **-**1.**,** **-**2.**,** **-**3.**])**

x\_eval **=** np**.**array**([**2.**,** 5.**,** 8.**,** 1.**])**

y\_eval **=** np**.**array**([-**1.01**,** **-**4.1**,** **-**7**,** 0.**])**

input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_train**},** y\_train**,** batch\_size**=**4**,** num\_epochs**=None,** shuffle**=True)**

train\_input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_train**},** y\_train**,** batch\_size**=**4**,** num\_epochs**=**1000**,** shuffle**=False)**

eval\_input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_eval**},** y\_eval**,** batch\_size**=**4**,** num\_epochs**=**1000**,** shuffle**=False)**

# We can invoke 1000 training steps by invoking the method and passing the

# training data set.

estimator**.**train**(**input\_fn**=**input\_fn**,** steps**=**1000**)**

# Here we evaluate how well our model did.

train\_metrics **=** estimator**.**evaluate**(**input\_fn**=**train\_input\_fn**)**

eval\_metrics **=** estimator**.**evaluate**(**input\_fn**=**eval\_input\_fn**)**

**print(**"train metrics: %r"**%** train\_metrics**)**

**print(**"eval metrics: %r"**%** eval\_metrics**)**

Le code est relativement simple. On peut noter que :

* on ne construit plus le graphe nous même
* On ne lance plus de run manuellement. Tout ceci est fait dans les fonctions train et evaluate.

## Modèle personnalisé avec TfEstimator

tf.estimator ne vous verrouille pas dans ses modèles prédéfinis. Supposons que nous voulions créer un modèle personnalisé qui n'est pas intégré dans TensorFlow, en conservant l'abstraction de haut niveau de jeu de données, l' alimentation, la formation, etc. de tf.estimator. Içi, nous allons implémenter notre propre modèle équivalent au LinearRegressor.

Pour définir un modèle personnalisé qui fonctionne avec tf.estimator, nous devons utiliser tf.estimator.Estimator.  Nous fournissons simplement à Estimator une fonction model\_fn qui indique à tf.estimator comment il peut évaluer les prédictions, les étapes d'entraînement et la perte. Le code est comme suit:

**import** numpy **as** np

**import** tensorflow **as** tf

# Declare list of features, we only have one real-valued feature

**def** ma\_fn**(**features**,** labels**,** mode**):**

# Build a linear model and predict values

W **=** tf**.**get\_variable**(**"W"**,** **[**1**],** dtype**=**tf**.**float64**)**

b **=** tf**.**get\_variable**(**"b"**,** **[**1**],** dtype**=**tf**.**float64**)**

y **=** W**\***features**[**'x'**]** **+** b

# Loss sub-graph

loss **=** tf**.**reduce\_sum**(**tf**.**square**(**y **-** labels**))**

# Training sub-graph

global\_step **=** tf**.**train**.**get\_global\_step**()**

optimizer **=** tf**.**train**.**GradientDescentOptimizer**(**0.01**)**

train **=** tf**.**group**(**optimizer**.**minimize**(**loss**),**

tf**.**assign\_add**(**global\_step**,** 1**))**

# EstimatorSpec connects subgraphs we built to the

# appropriate functionality.

**return** tf**.**estimator**.**EstimatorSpec**(**

mode**=**mode**,**

predictions**=**y**,**

loss**=**loss**,**

train\_op**=**train**)**

estimator **=** tf**.**estimator**.**Estimator**(**model\_fn**=**ma\_fn**)**

# define our data sets

x\_train **=** np**.**array**([**1.**,** 2.**,** 3.**,** 4.**])**

y\_train **=** np**.**array**([**0.**,** **-**1.**,** **-**2.**,** **-**3.**])**

x\_eval **=** np**.**array**([**2.**,** 5.**,** 8.**,** 1.**])**

y\_eval **=** np**.**array**([-**1.01**,** **-**4.1**,** **-**7.**,** 0.**])**

input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_train**},** y\_train**,** batch\_size**=**4**,** num\_epochs**=None,** shuffle**=True)**

train\_input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_train**},** y\_train**,** batch\_size**=**4**,** num\_epochs**=**1000**,** shuffle**=False)**

eval\_input\_fn **=** tf**.**estimator**.**inputs**.**numpy\_input\_fn**(**

**{**"x"**:** x\_eval**},** y\_eval**,** batch\_size**=**4**,** num\_epochs**=**1000**,** shuffle**=False)**

# train

estimator**.**train**(**input\_fn**=**input\_fn**,** steps**=**1000**)**

# Here we evaluate how well our model did.

train\_metrics **=** estimator**.**evaluate**(**input\_fn**=**train\_input\_fn**)**

eval\_metrics **=** estimator**.**evaluate**(**input\_fn**=**eval\_input\_fn**)**

**print(**"train metrics: %r"**%** train\_metrics**)**

**print(**"eval metrics: %r"**%** eval\_metrics**)**

## Utiliser Tf.Estimator :

Nous allons utiliser tf.estimator pour construire un classificateur de [réseau neuronal](https://en.wikipedia.org/wiki/Artificial_neural_network) et l'entraîner sur l' [ensemble de données Iris](https://en.wikipedia.org/wiki/Iris_flower_data_set) 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](http://download.tensorflow.org/data/iris_training.csv) )
* Un ensemble de test de 30 échantillons ( [iris\_test.csv](http://download.tensorflow.org/data/iris_test.csv) ).

Voici les codes pour construire le code complet du classificateur de réseau neuronal:

**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** tensorflow **as** tf

# Data sets

IRIS\_TRAINING **=** "iris\_training.csv"

IRIS\_TRAINING\_URL **=** "http://download.tensorflow.org/data/iris\_training.csv"

IRIS\_TEST **=** "iris\_test.csv"

IRIS\_TEST\_URL **=** "http://download.tensorflow.org/data/iris\_test.csv"

**def** main**():**

# 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**)**

ce code ci dessus importez tous les modules nécessaires et définissez où télécharger et stocker l'ensemble de données et il télécharge les ensembles de formation et de test s' ils ne sont pas stocker localement.

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**)**

ce code ci dessus chargez les ensembles de formation en utilisant la fonction **training\_set** et les ensembles de test utilisant la fonction **test\_set.**

feature\_columns **=** **[**tf**.**feature\_column**.**numeric\_column**(**"x"**,** shape**=[**4**])]**

cette fonction définit les colonnes d'entités du modèle et construire les colonnes d'entités

classifier **=** tf**.**estimator**.**DNNClassifier**(**feature\_columns**=**feature\_columns**,**

hidden\_units**=[**10**,** 20**,** 10**],**

n\_classes**=**3**,**

model\_dir**=**"/tmp/iris\_model"**)**

cette fonction crée un DNN classifier modèle ,  qui respecte les arguments suivant:

* L'ensemble des colonnes de caractéristiques .
* Trois [couches cachées](http://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw) , contenant 10, 20 et 10 neurones, respectivement.
* Trois classes cibles, représentant les trois espèces d'Iris.
* Le répertoire dans lequel TensorFlow sauvegardera les données de point de contrôle et les résumés TensorBoard pendant l'apprentissage du modèle.

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)**

ce code produit le pipeline d'entrée.

classifier**.**train**(**input\_fn**=**train\_input\_fn**,** steps**=**2000**)**

cette fonction train et refait le train 2000 fois.?????????????????????????????

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)**

ce code définie les entrées teste.

accuracy\_score **=** classifier**.**evaluate**(**input\_fn**=**test\_input\_fn**)[**"accuracy"**]**

**print(**"\nTest Accuracy: {0:f}\n"**.**format**(**accuracy\_score**))**

ce code évalue les **test \_set.data** et le **test\_set.traget** et affiche le résulta .

la sortie de ce programme serai:

Learning Accuracy: 1.000

Test Accuracy: 0.966667

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 **=** list**(**classifier**.**predict**(**input\_fn**=**predict\_input\_fn**))**

**for** p **in** predictions **:**

chaine **=**p**[**"classes"**]**

**print** **(**"classe "**,** chaine**[**0**].**decode**())**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

main**()**

ce programme récupère affiche le résultat et les class .

la sortie de ce programme (le programme complet ) serai:

Learning Accuracy: 1.000000

Test Accuracy: 0.966667

classe 1

classe 2