



APIs haut niveau de TensorFlow

Hands-On - Devovx 2019



Yoann Benoit
Data Scientist
@YoannBENOIT



Giulia Bianchi
Data Scientist
@Giuliabianchl



Commençons par le commencement

Un bon vieux code TensorFlow bas niveau

Caractéristiques de TensorFlow

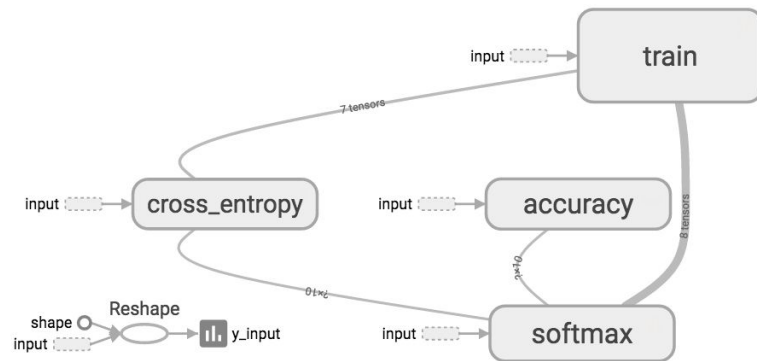
- Représentation des calculs sous forme de **graphe d'exécution**
- Chaque noeud est une opération, et chaque lien est un tenseur (une matrice multidimensionnelle)

```
import tensorflow as tf

matrix1 = tf.constant([[3., 3.]])
matrix2 = tf.constant([[2.], [2.]])

product = tf.matmul(matrix1, matrix2)

with tf.Session() as sess:
    result = sess.run(product)
    print(result)
```



Couche dense avec l'API Bas Niveau

L'utilisation de l'API bas niveau de TensorFlow oblige à déclarer chacune des variables et écrire chaque opération matricielle.

Parfois nécessaire, souvent too much !

```
with tf.name_scope(name_scope):  
  
    weights = tf.Variable(tf.truncated_normal(shape=[num_neurons_previous_layer,  
                                                    num_neurons_current_layer],  
                                                    stddev=0.1, name="weights"))  
  
    biases = tf.Variable(tf.constant(0.1, shape=[num_neurons_current_layer],  
                                       name="biases"))  
  
    relu = tf.nn.relu(tf.matmul(x, weights) + biases, name=name_scope)
```

} Utilisation de
name_scope pour
regrouper plusieurs
éléments d'un graphe

} Définition des
poids et biais
avec *tf.Variable*

} Ecriture du
produit matriciel

Couche convolutionnelle avec l'API Bas Niveau

```
with tf.name_scope(name_scope):  
  
    weights = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 16],  
                                              stddev=0.1, name="weights"))  
  
    biases = tf.Variable(tf.constant(0.1, shape=[num_channels_in_current_layer],  
                                       name="biases"))  
  
    relu = tf.nn.relu(tf.nn.conv2d(x,  
                                   weights,  
                                   strides=[1, 1, 1, 1],  
                                   padding="SAME") + biases,  
                       name=name_scope)
```

Utilisation de
`tf.nn.conv2d`



Trop long !



TensorFlow 2.0, ça ressemble à quoi ?

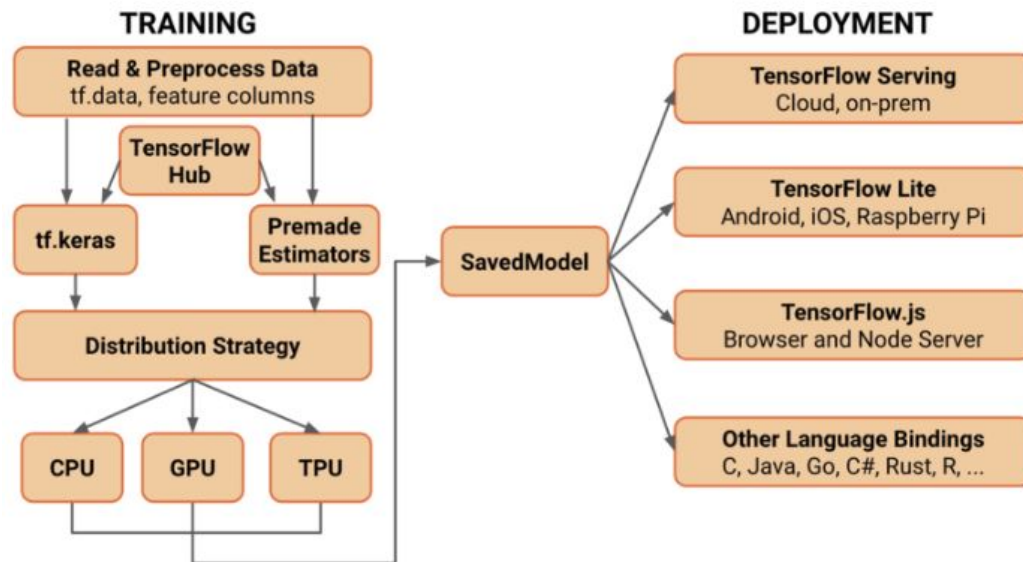
Tour d'horizon

Aperçu de TensorFlow 2.0

TensorFlow 2.0 se focalisera sur la simplicité d'utilisation :

- Construction de modèles via **tf.keras** et **eager execution**
- Déploiement robuste de modèle sur tout type de **plateforme**
- Facilitation de **l'expérimentation**
- Simplification des **APIs** et cleaning des anciennes versions

Un framework homogène permettant de créer des workflows de Machine Learning du prototypage au déploiement.



Construction de modèles simplifiée

Chargement des données avec
tf.data

Construction de pipelines
de données d'entrée avec
tf.data

Gestion du feature
engineering avec
tf.feature_columns

Gestion de donnée en
mémoire ou non

Construction de modèles simplifiée

Chargement des données avec
tf.data

Construction, entraînement et
validation de modèles avec
tf.keras ou les **Premade
Estimators**

Construction de pipelines
de données d'entrée avec
tf.data

Gestion du feature
engineering avec
tf.feature_columns

Gestion de donnée en
mémoire ou non

Intégration de **Keras** avec
le reste de l'écosystème
TensorFlow

Utilisation d'un set de
modèles pré-packagés
(**Premade Estimators**)

Transfer Learning via
tf.hub

Construction de modèles simplifiée

Chargement des données avec
tf.data

Construction, entraînement et
validation de modèles avec
tf.keras ou les **Premade
Estimators**

Run & Debug avec la **eager
execution**, puis bénéficier des
avantages des graphes via
tf.function

Construction de pipelines
de données d'entrée avec
tf.data

Gestion du feature
engineering avec
tf.feature_columns

Gestion de donnée en
mémoire ou non

Intégration de **Keras** avec
le reste de l'écosystème
TensorFlow

Utilisation d'un set de
modèles pré-packagés
(**Premade Estimators**)

Transfer Learning via
tf.hub

Eager Execution sera le
mode par défaut dans
TensorFlow 2.0

tf.function permettra de
transposer de manière
transparente le code en
graphes TensorFlow pour
bénéficier de leurs
optimisations de
performances

Construction de modèles simplifiée

Chargement des données avec
tf.data

Construction, entraînement et
validation de modèles avec
tf.keras ou les **Premade
Estimators**

Run & Debug avec la **eager
execution**, puis bénéficier des
avantages des graphes via
tf.function

Utilisation des **distribution
strategies** pour l'entraînement
distribué

Construction de pipelines
de données d'entrée avec
tf.data

Gestion du feature
engineering avec
tf.feature_columns

Gestion de donnée en
mémoire ou non

Intégration de **Keras** avec
le reste de l'écosystème
TensorFlow

Utilisation d'un set de
modèles pré-packagés
(**Premade Estimators**)

Transfer Learning via
tf.hub

Eager Execution sera le
mode par défaut dans
TensorFlow 2.0

tf.function permettra de
transposer de manière
transparente le code en
graphes TensorFlow pour
bénéficier de leurs
optimisations de
performances

L'API **Distribution
Strategy** permet de
distribuer l'entraînement
de modèles sur différentes
configuration de
Hardware sans changer la
définition du modèle.

Construction de modèles simplifiée

Chargement des données avec
tf.data

Construction, entraînement et
validation de modèles avec
tf.keras ou les **Premade
Estimators**

Run & Debug avec la **eager
execution**, puis bénéficier des
avantages des graphes via
tf.function

Utilisation des **distribution
strategies** pour l'entraînement
distribué

Export en **SavedModel**

Construction de pipelines
de données d'entrée avec
tf.data

Gestion du feature
engineering avec
tf.feature_columns

Gestion de donnée en
mémoire ou non

Intégration de **Keras** avec
le reste de l'écosystème
TensorFlow

Utilisation d'un set de
modèles pré-packagés
(**Premade Estimators**)

Transfer Learning via
tf.hub

Eager Execution sera le
mode par défaut dans
TensorFlow 2.0

tf.function permettra de
transposer de manière
transparente le code en
graphes TensorFlow pour
bénéficier de leurs
optimisations de
performances

L'**API Distribution
Strategy** permet de
distribuer l'entraînement
de modèles sur différentes
configuration de
Hardware sans changer la
définition du modèle.

SavedModel devient de
format de sérialisation
standardisé pour
TensorFlow Serving,
TensorFlow Lite,
TensorFlow.js,
TensorFlow Hub, etc.

Déploiement sur tout type de plateforme

TensorFlow 2.0 améliore la compatibilité entre les différentes plateformes sur lesquelles on cherche à déployer des modèles en standardisant les formats d'échange et en homogénéisant les APIs

TensorFlow Serving

Serving des modèles via HTTP / REST ou gRPC / Protocol Buffers.

TensorFlow Lite

Version allégée de TensorFlow pour usage sur des devices mobile ou des systèmes embarqués légers. Permet le déploiement de modèles sur Android, iOS, Raspberry Pi ou Edge TPU.

TensorFlow.js

Déploiement de modèles dans des environnements Javascript (sur le browser ou côté server via Node.js). Permet aussi l'entraînement de modèles sur le browser.

Sans oublier le support d'autres langages comme C, Java, Go, Rust, Julia, R, etc.

Ca reste pas très “Python-Friendly” tout ça ...

Aperçu de la eager execution

C'est quoi le problème ?

TensorFlow est initialement un **moteur d'exécution orienté graphe**

- C'est ce qui lui a souvent été reproché par les utilisateurs
- Oblige à définir toutes leurs opérations avant de les exécuter dans un graphe
- Peu commun dans le monde du développement Python

Pourquoi avoir fait ce choix ?

- **Différenciation automatique** des opérations
- Possibilité de **déployer** sur un serveur n'exécutant pas du Python, ou sur un smartphone
- **Optimisations** au niveau du graphe : vue complète du graphe d'exécution requise pour trouver des optimisations globales
- Distribution "automatique" sur des centaines de machines

Qu'est-ce que la *eager execution* ?

Avantages à avoir un mode de fonctionnement plus “Python-Friendly” :

- Facilité pour débbugger et pour utiliser des **outils d'analyse du code**
- Une manière plus **dynamique** de construire son code
- Possibilité d'itérer beaucoup plus rapidement (tester son modèle lors de sa construction, rajouter des opérations facilement, etc.).

NB : Peut cependant être plus lent que l'exécution en mode graphe

```
import tensorflow as tf

tf.enable_eager_execution()

a = tf.constant([[2.0, 3.0], [4.0, 5.0]])
print(tf.matmul(a, a))

# => tf.Tensor([[16., 21.], [28., 37.]], shape=(2,2), dtype=float32)
```

Hands-on TensorFlow 2.0

Manipulations basiques

Hands-on TensorFlow 2.0 - Les bases

Hands on tensorflow high level apis

Tutorial on TensorFlow High Level APIs

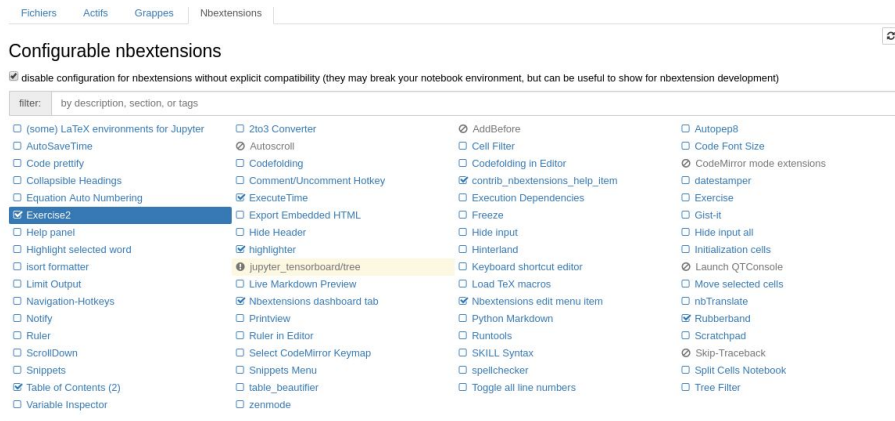
 launch 

Extensions to add

The notebooks work with jupyter notebook extensions. Once you access jupyter, before starting the notebook, go on the **Nbextensions** tab and check the following extensions:

- `Exercise2`
- `ExecuteTime`
- `highlighter`
- `Table of Contents (2)`

You can then start the notebooks, and you're good to go!



Project URL : https://github.com/xebia-france/hands_on_tensorflow_high_level_apis

3 available environments :

- Virtualenv
- Docker
- Binder (*in case you have neither virtualenv nor Docker*)

Hands-on TensorFlow 2.0 - Les bases

TensorFlow se base sur la manipulation de *tenseurs*.

Un tenseur est un objet mathématique à plusieurs dimensions :

- Un vecteur est un tenseur à 1 dimension
- Une matrice est un tenseur à 2 dimensions
- Une image RGB est un tenseur à 3 dimensions

Les opérations mathématiques entre tensors sont la base du deep learning.

Hands-on TensorFlow 2.0 - Les bases

```
import tensorflow as tf

tf.constant([1, 2, 3])
# <tf.Tensor: id=0, shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>

tns_1 = tf.constant([[1, 2, 3], [4, 5, 6]])
tns_2 = tf.constant([[4, 5, 6], [1, 2, 3]])

tf.matmul(tns_1, tf.transpose(tns_2))
# <tf.Tensor: id=4, shape=(2, 2), dtype=int32, numpy=array([[32, 14], [77, 32]], dtype=int32)>
```

En conclusion

Eager Execution

Pour la recherche, le prototypage, le débogage

Graph Mode

Pour la production et l'entraînement sur les gros volumes de données

Possibilité de passer d'un code écrit en eager mode à un code orienté graphe grâce à **tf.function** et **Autograph**.

Pas besoin de recoder chaque opération

Introduction à `tf.keras`

Qu'est-ce que Keras ?

Framework à part entière pour le Deep Learning

- **User Friendly** : Une interface simple et homogène, adaptée pour une grande majorité de Use Cases
- **Modulaire** : Tout est fait pour pouvoir assembler et composer les différentes briques d'un modèle et gérer son entraînement
- **Extensible** : Possibilité d'écrire ses propres *layers* ou fonctions de coût pour prototyper de nouvelles idées
- **Pour tous les niveaux** : Pour découvrir le Deep Learning aussi bien que pour les expérimentés du sujet

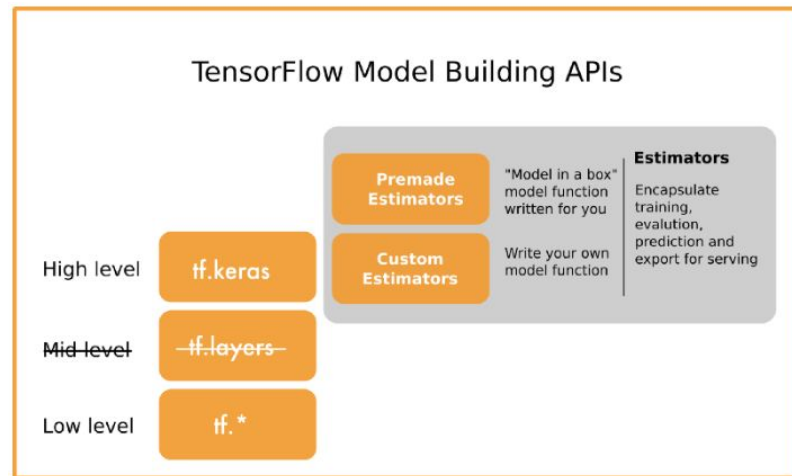
Disponible indépendamment ou intégré à TensorFlow avec **tf.keras**

Pourquoi utiliser tf.keras et non pas juste Keras ?

tf.keras permet nativement une meilleure intégration avec les autres composants de TensorFlow, notamment :

- Fonctionnement avec la **eager execution** (itérations immédiates et débogage simplifié)
- Compatibilité avec **tf.data** (pour gérer des pipelines scalables de données d'entrée)
- Compatibilité avec la notion d'**estimator**

L'avantage indéniable consiste à utiliser un même framework de bout en bout.



Utilisation de tf.keras - API Sequential

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation="relu"),
    tf.keras.layers.Dropout(0.2)
    tf.keras.layers.Dense(10, activation="softmax")
])

model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Certains datasets sont téléchargeables directement dans l'API

Utilisation des layers Flatten, Dense et Dropout

Compilation du modèle avec le type de loss, d'optimiser et de métrique à calculer

Entraînement et évaluation du modèle

Utilisation de tf.keras - API Fonctionnelle

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

inputs = tf.keras.layers.Input(shape=(28,28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(512, activation="relu")(x)
x = tf.keras.layers.Dropout(0.2)(x)
predictions = tf.keras.layers.Dense(10, activation="softmax")(x)

model = tf.keras.Model(inputs=inputs, outputs=predictions)

model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Certains datasets sont téléchargeables directement dans l'API

Utilisation des layers Flatten, Dense et Dropout

Compilation du modèle avec le type de loss, d'optimiser et de métrique à calculer

Entraînement et évaluation du modèle

Hands-on TensorFlow 2.0

tf.keras

Quand revenir vers du code bas niveau ?

- Pour des usages de recherche
- Pour un contrôle total et détaillé de chaque élément de votre expérience
- Pour implémenter des éléments très spécifiques et custom
- Pour étudier le Deep Learning et implémenter chaque étape

La vie, c'est pas que des jolis datasets

Introduction à tf.data

Qu'est-ce que tf.data ?

- API permettant de construire des **pipelines complexes de données d'entrée**, découpés en morceaux simples et réutilisables
- Permet de gérer de la **lecture** de données sur disque à la gestion du **shuffle** et des **batch**, en passant par les **transformations** intermédiaires de la donnée

Fonctionnement de tf.data

1. Construire un **Dataset**. Ex: **tf.data.Dataset.from_tensor_slices()** ou **tf.data.TFRecordDataset**
2. Le transformer en un autre Dataset en chaînant des appels de de fonction sur l'objet initial
 - a. Via des méthodes comme **map** pour des applications de fonctions élément par élément par exemple
 - b. Via des méthodes comme **batch** pour appliquer des transformations multi-éléments
3. Consommer le Dataset via un **Iterator** qui permet d'accéder aux éléments du dataset les uns après les autres

Création de dataset

Via un array numpy en mémoire

```
features, labels = (np.random.sample((100,2)), np.random.sample((100,1)))  
dataset = tf.data.Dataset.from_tensor_slices((features,labels))
```

Depuis des données au format TFRecords

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]  
dataset = tf.data.TFRecordDataset(filenames)
```

Création de dataset

Depuis des fichiers texte

```
filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]  
dataset = tf.data.TextLineDataset(filenames)
```

Depuis des fichiers csv

```
filenames = ["/var/data/file1.csv", "/var/data/file2.csv"]  
record_defaults = [[0.0]] * 2 # Only provide defaults for the selected columns  
dataset = tf.contrib.data.CsvDataset(filenames, record_defaults, header=True, select_cols=[2,4])
```

Preprocessing des données avec Dataset.map()

- Transformation produisant un nouveau Dataset en **appliquant une fonction f à chaque élément** du dataset d'entrée
- La fonction f prend en entrée un **tf.Tensor** (représentant un élément du dataset d'entrée) et retourne un autre **tf.Tensor** (représentant un élément du nouveau dataset)

Preprocessing des données avec Dataset.map()

```
def _extract_fn(tfreord):  
    features = {  
        'height': tf.FixedLenFeature([], tf.int64), 'width': tf.FixedLenFeature([], tf.int64),  
        'image_raw': tf.FixedLenFeature([], tf.string), 'class_id_raw': tf.FixedLenFeature([], tf.int64)  
    }  
    return tf.parse_single_example(tfreord, features)  
  
dataset = tf.data.TFRecordDataset(filenamees)  
dataset = dataset.map(_extract_fn)
```

Preprocessing des données avec Dataset.map()

```
def _train_preprocess(image, label):  
    image = tf.image.random_flip_left_right(image)  
  
    image = tf.image.random_brightness(image, max_delta=32.0 / 255.0)  
    image = tf.image.random_saturation(image, lower=0.5, upper=1.5)  
  
    image = tf.clip_by_value(image, 0.0, 1.0)  
  
    return image, label  
  
dataset = tf.data.TFRecordDataset(filenamees)  
dataset = dataset.map(_extract_fn)  
dataset = dataset.map(_train_preprocess)
```

Création d'un workflow d'entraînement

Créer des batches

- Créer des batches revient à stacker n éléments consécutifs du Dataset en un seul élément
- La transformation se fait via **Dataset.batch()**

Préparer la donnée pour entraîner sur plusieurs *epochs*

- Une *epoch* constitue une passe complète sur le dataset
- La transformation se fait via **Dataset.repeat()**
- Ne permet pas de signaler la fin d'une epoch

```
dataset = (dataset
            .map(_extract_fn)
            .map(_train_preprocess)
            .shuffle(10000)
            .repeat(NUM_EPOCHS)
            .batch(BATCH_SIZE))
```

Shuffle de la donnée

- Permet de réordonner aléatoirement la donnée
- La transformation se fait via **Dataset.shuffle(buffer_size)**

Quelques notes sur la performance

- Utilisation de GPUs ou TPUs -> Pipeline de données d'entrée peut devenir un **goulot d'étranglement** !
- Un bon pipeline doit avoir préparé le prochain step avant que le précédent ne soit terminé

Extract

- Lecture depuis un système de stockage local ou remote

Transform

- Utilisation des coeurs CPU pour faire le preprocessing de la donnée

Load

- Chargement des données transformées sur les devices qui accélèrent les calculs (GPUs, TPUs)

Zoom sur la performance - Prefetch

Sans
`tf.data.Dataset.prefetch()`

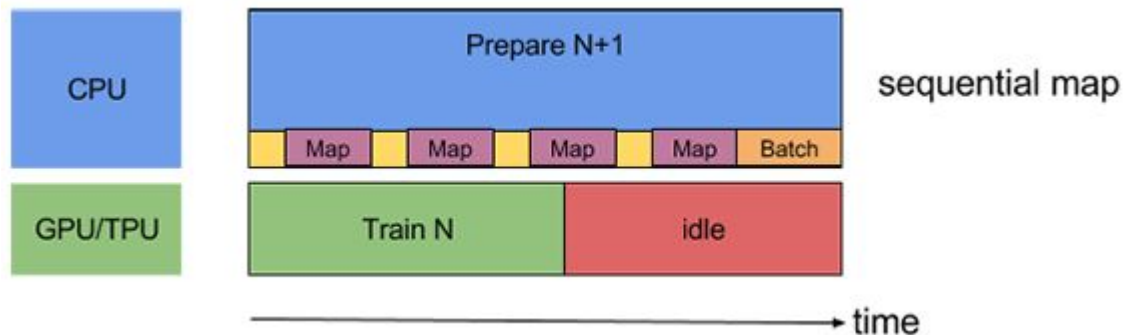


Avec
`tf.data.Dataset.prefetch()`

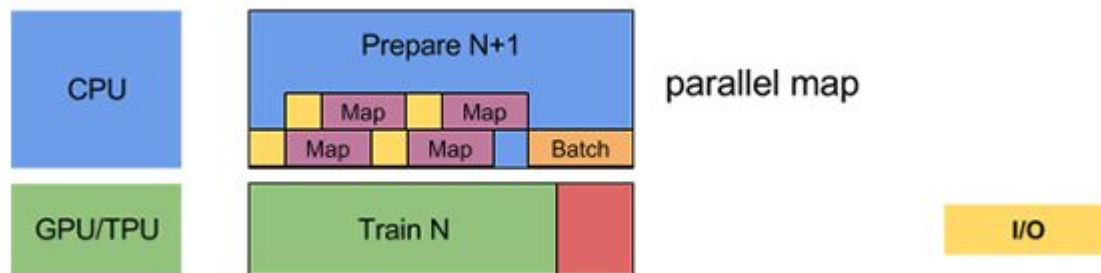


Zoom sur la performance - Parallélisation

Sans *num_parallel_calls*
en argument du map



Avec *num_parallel_calls=2*
en argument du map



Appel au fit du modèle

L'appel au fit diffère légèrement avec `tf.data` :

- Le dataset doit contenir à la fois l'information pour les données ainsi que pour les labels
- Il n'y a plus besoin de préciser la taille de batch, mais plutôt le nombre de steps souhaité par epoch

```
# Fit sur des arrays numpy
model.fit(train_images, train_labels, epochs=5, batch_size=32)
```

```
# Fit sur des Dataset
model.fit(dataset, epochs=5, steps_per_epoch=1000)
```

Hands-on TensorFlow 2.0

tf.data

Take Away - Pourquoi utiliser tf.data ?

- Gestion optimisée des pipelines de données d'entrée
- Pour la lecture et du preprocessing
- Compatible avec tout l'écosystème TensorFlow
- Du prototype à la production avec le même code

Pourquoi repartir from scratch à chaque fois ?

Transfer Learning avec TensorFlow Hub

Transfer Learning avec tf.keras

```
conv_base = tf.keras.applications.NASNetMobile(weights='imagenet',  
                                                include_top=False,  
                                                input_shape=(224, 224, 3))
```

```
model = tf.keras.models.Sequential()  
model.add(conv_base)  
model.add(tf.keras.layers.Flatten())  
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'))
```

```
conv_base.trainable = False
```

Keras met à disposition un ensemble de modèles pré-entraînés utilisables tels quels

Ajout d'une couche Dense à la base convolutionnelle

Rendre les poids de la base convolutionnelle non modifiables pendant l'entraînement

TensorFlow Hub

- Librairie pour publier, découvrir et consommer des parties réutilisables de modèles de Machine Learning entraînés sur des datasets éprouvés (**Model Zoo**)
- Fonctionne sous forme de **module**
 - Morceau indépendant de **graphe TensorFlow**
 - Vient avec tous les poids
- Les modules sont utilisés pour des usages de **Transfer Learning**
 - Entraîner un modèle profond lorsque l'on a peu de données
 - Améliorer les capacités de généralisation d'un modèle
 - Accélérer l'entraînement des modèles
 - Faire du fine-tuning pour certains modules
- Anciennement [TF-Slim](#), avec beaucoup plus de difficultés pour son utilisation

- Text
- Embedding
- Image
- Classification
- Feature Vector
- Generator
- Other
- Video
- Classification
- Publishers
- Google
- DeepMind

Text embedding

- universal-sentence-encoder

By Google

text-embedding DAN English

Encoder of greater-than-word length text trained on a variety of data.
- universal-sentence-encoder-large

By Google

text-embedding Transformer English

Encoder of greater-than-word length text trained on a variety of data.
- elmo

By Google

text-embedding 1 Billion Word Benchmark ELMo English

Embeddings from a language model trained on the 1 Billion Word Benchmark.

[View more text embeddings](#)

Image feature vectors

- imagenet/inception_v3/feature_vector

By Google

image-feature-vector ImageNet (ILSVRC-2012-CLS) Inception V3

Feature vectors of images with Inception V3 trained on ImageNet (ILSVRC-2012-CLS).
- imagenet/mobilenet_v2_100_224/feature_vector

By Google

image-feature-vector ImageNet (ILSVRC-2012-CLS) MobileNet V2

Feature vectors of images with MobileNet V2 (depth multiplier 1.00) trained on ImageNet (ILSVRC-2012-CLS).
- imagenet/mobilenet_v2_140_224/feature_vector

By Google

image-feature-vector ImageNet (ILSVRC-2012-CLS) MobileNet V2

Feature vectors of images with MobileNet V2 (depth multiplier 1.40) trained on ImageNet (ILSVRC-2012-CLS).

Utilisation de TensorFlow Hub

```
import tensorflow_hub as hub
```

```
module = hub.Module('https://tfhub.dev/google/imagenet/inception_v3/classification/1')
```

Un module peut posséder plusieurs signatures et possède ses propres caractéristiques en termes d'input et output

```
print(module.get_signature_names())  
# ['default', 'image_classification', 'image_feature_vector']
```

```
print(module.get_input_info_dict()) # When no signature is given, considers it as 'default'  
# {'images': <hub.ParsedTensorInfo shape=(?, 299, 299, 3) dtype=float32 is_sparse=False>}
```

```
print(module.get_output_info_dict()) # When no signature is given, considers it as 'default'  
# {'default': <hub.ParsedTensorInfo shape=(?, 1001) dtype=float32 is_sparse=False>}
```

Exemples de modules

Image Classification

- Le module contient toute l'architecture du réseau de neurones jusqu'à la prédiction de probabilité d'appartenance aux différentes classes

Feature Vector

- Similaire au module de classification d'image, auquel on a retiré la couche dense finale
- Spécifiquement utilisé pour le Transfer Learning

Classification de vidéos

- Pour la catégorisation automatique des différentes séquences d'une vidéo

Exemples de modules

Text embedding

- Embedding dans plusieurs langues
- Fait à l'échelle de la phrase pour la majorité, pas forcément au niveau du mot

Object detection

- Détection et classification automatisée d'objets sur une image
- Ne permet pas de faire du fine-tuning pour le moment

Modules génératifs

- Generative Adversarial Networks pré-entraînés pour générer de nouvelles images
- Les modules n'exposent que les Generators, pas les Discriminators

Transfer Learning avec TensorFlow Hub et tf.keras

```
import tensorflow_hub as hub

feature_extractor_url =
    "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/2"

model = tf.keras.Sequential([
    hub.KerasLayer(conv_base_url, output_shape=1280, trainable=False),
    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])

model.build([None, 224, 224])
```

URL du module de feature extraction

Utilisation de KerasLayer pour utiliser le module dans un modèle Keras

Initialisation et ajout d'informations sur la taille des batchs d'entrée

Hands-on TensorFlow 2.0

Tensorflow Hub

Take Away - Les avantages de TensorFlow Hub

- Large éventail de modèles pré-entraînés
- Compatible avec tout l'écosystème TensorFlow
- Intégration simplifiée avec tf.keras et les estimators
- *Warning - TensorFlow 2.0 est toujours en alpha, pour le moment tout n'est pas encore compatible*

Et comment on encapsule tout ça ?

Aperçu de l'API Estimators

Qu'est-ce qu'un Estimator ?

- Autre API haut niveau de TensorFlow pour simplifier les workflows de ML
- Encapsule les steps de training, évaluation, prédiction et export pour le serving

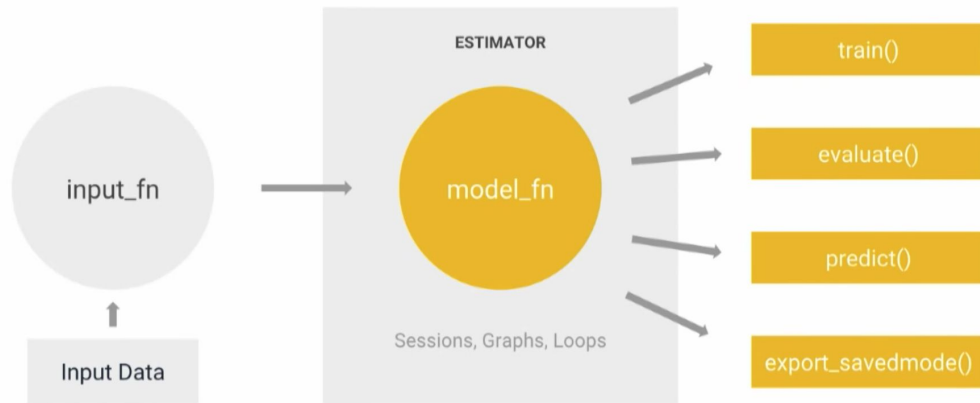
Deux types d'Estimator :

- **Premade Estimators** : la *model function* est déjà écrite pour vous. C'est un modèle pré-packagé pour vous qu'il ne reste plus qu'à entraîner
- **Custom Estimators** : A vous d'écrire votre propre *model function*

A quoi sert un estimator ?

- Possibilité de passer d'un environnement local CPU à un environnement multi-GPU ou TPU sans avoir à changer le code du modèle
- Interface unifiée pour le partage de code
- Ils sont basés sur l'API **tf.keras**
- Pas besoin de gérer la construction du graphe, il le fait pour vous

La gestion des données d'entrée se fait séparément afin de faciliter l'expérimentation avec plusieurs datasets



Premade Estimators

- Ils gèrent les détails d'implémentation pour nous afin de se concentrer sur l'expérience à mener
- Les valeurs par défaut raisonnables et utilisables dans de nombreux contextes
- Le même Estimator est utilisé quel que soit le nombre et le type de features à utiliser
- Implémentés avec les best practices pour optimiser les performances

Fonctionnement

1. Créer une fonction d'import de data
2. Définir ses **Feature Columns**
3. Instancier un **Premade Estimator**
4. Appel aux méthodes **train**, **evaluate** et **inference**

Feature Columns

Facilitent l'expérimentation avec les features des datasets.
Définissent comment nous souhaitons traiter la donnée pour le modèle, avec un minimum de code.

Données numériques

```
tf.feature_column.numeric_column(key="SepalLength")
```

Bucketizing

```
tf.feature_column.bucketized_column(source_column="year", boundaries=[1960, 1980, 2000])
```

Catégories avec vocabulaire

```
tf.feature_column.categorical_column_with_vocabulary_list(  
    key=feature_name_from_input_fn,  
    vocabulary_list=["kitchenware", "electronics", "sports"])
```

Embedding

```
tf.feature_column.embedding_column(  
    categorical_column=categorical_column,  
    dimension=embedding_dimensions)
```

Premade Estimators

DNNClassifier
DNNRegressor

Création de réseaux de neurones profonds pour la classification ou la régression

LinearClassifier
LinearRegressor

Modèles linéaires simples pour la classification ou la régression

DNNLinearCombinedClassifier
DNNLinearCombinedRegressor

Combinaison d'un DNN et d'un modèle linéaire (Wide & Deep Learning)

BoostedTreesClassifier
BoostedTreesRegressor

Modèles de Gradient Boosted Trees pour la classification ou la régression

Création d'un DNNClassifier

```
estimator = DNNClassifier(hidden_units=[1024, 256, ...],  
                           feature_columns=feature_columns)
```

} *Instanciation du Premade Estimator*

```
train_spec = TrainSpec(input_fn=lambda: input_fn(TRAIN_FILES), max_steps=1000)  
eval_spec = EvalSpec(input_fn=lambda: input_fn(EVAL_FILES))
```

} *Création des Specs pour l'entraînement et la validation*

```
train_and_evaluate(estimator, train_spec, eval_spec)
```

} *Lancement de l'entraînement et de l'évaluation*

Custom Estimators

```
return tf.estimator.Estimator(  
    model_fn=model_fn,  
    config=config,  
    params=params,  
)
```

- **model_fn** sert à initialiser le modèle. Défini par un **EstimatorSpec** qui définit comment entraîner et évaluer le modèle.
- **config** est un objet **RunConfig** qui spécifie comment faire tourner le modèle (checkpoints, stratégie de distribution, etc.).
- **params** contient les hyperparamètres du modèle.

Encore plus simple avec tf.keras : model_to_estimator

```
BATCH_SIZE = 64
```

```
EPOCHS = 5
```

```
estimator = tf.keras.estimator.model_to_estimator(model)
```

```
def input_fn(images, labels, epochs, batch_size):
```

```
    ds = tf.data.Dataset.from_tensor_slices((images, labels))
```

```
    ds = ds.shuffle(5000).repeat(epochs).batch(batch_size).prefetch(2)
```

```
    return ds
```

```
estimator.train(lambda: input_fn(train_images, train_labels,  
                                epochs=EPOCHS, batch_size=BATCH_SIZE))
```

```
estimator.evaluate(lambda: input_fn(test_images, test_labels,  
                                   epochs=1, batch_size=BATCH_SIZE))
```

} *Conversion du modèle Keras en Estimator*

} *Création d'une fonction d'input*

} *Entraînement du modèle*

} *Evaluation du modèle*

Hands-on TensorFlow 2.0

`tf.estimator`

Take Away - tf.keras ? Estimators ? Que choisir ?

- En général, rester avec **tf.keras** est amplement suffisant pour travailler du prototype à la production.
 - Possible d'exporter un modèle Keras directement dans le format SavedModel
- A moins d'utiliser les **Premade Estimators** qui sont utilisables clé en main
- A moins de travailler sur des infrastructures qui requièrent des Estimators
- Passer par **model_to_estimator** prioritairement

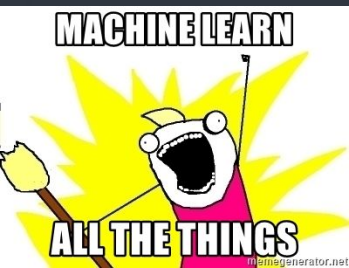


Conclusion

Take Away

TensorFlow était au début un framework pour le Deep Learning.

C'est maintenant un écosystème pour tous types d'applications de Machine Learning, sur toutes les étapes d'un workflow, pour tous types d'utilisateurs et sur tous types de plateformes.



Merci

The top corners of the image feature decorative clusters of small squares. In the top-left, there are orange and light brown squares. In the top-right, there are light purple and lavender squares. These clusters are arranged in a way that suggests a digital or data-related theme.

DATA X DAY

THE DATA CENTRIC CONFERENCE IN PARIS

A horizontal band of decorative squares runs across the middle of the image, just above the orange-to-purple gradient bar. This band consists of many small squares in various shades of orange, brown, and purple, arranged in a pattern that resembles a stylized city skyline or a data visualization.

JUNE, 27th 2019 - PAN PIPER, PARIS

DATAXDAY.FR