Taller de Aprendizaje Automático

Modelos y entrenamiento personalizado

Lectura de datos

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

Montevideo, 2024

1. Visión general de TensorFlow

2. Modelos y entrenamiento personalizado
   - Función de costo personalizada
   - Medida de desempeño personalizada
   - Cálculo de gradientes con autodiff
   - Entrenamiento personalizado

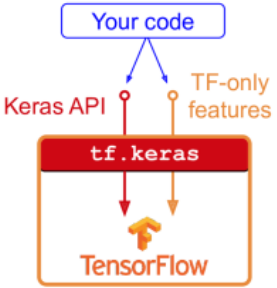
3. Data API

4. TF Records

Keras y TensorFlow

Keras API de alto nivel para aprendizaje profundo

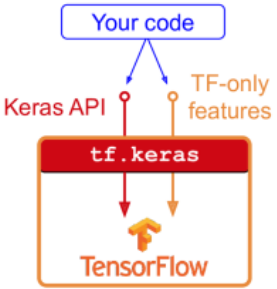
TensorFlow biblioteca para aprendizaje profundo



Keras y TensorFlow

Keras API de alto nivel para aprendizaje profundo

TensorFlow biblioteca para aprendizaje profundo




```
>>> import tensorflow as tf
>>> from tensorflow import keras
```

API de Keras

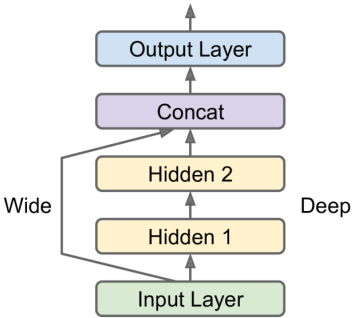
API Secuencial

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

API de Keras

API Secuencial

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```



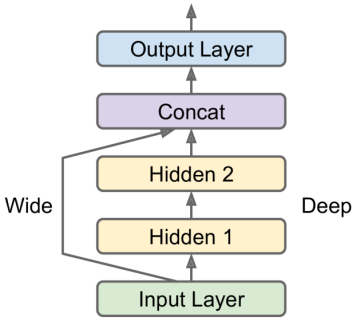
API de Keras

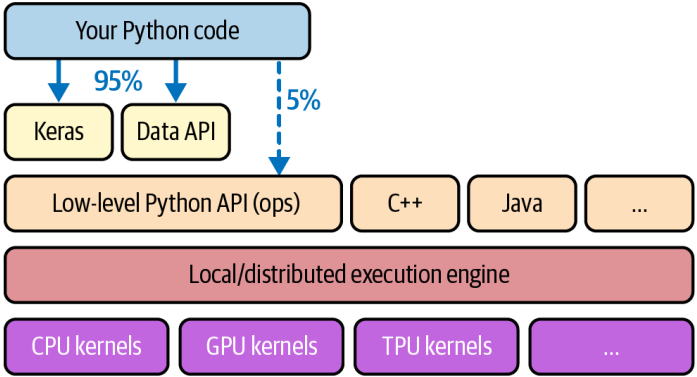
API Secuencial

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

API Funcional

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```





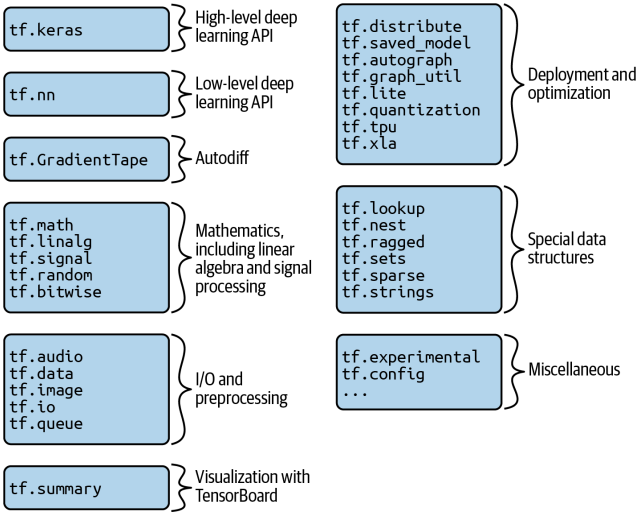
Arquitectura de TensorFlow [†]

[†] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022

Principales características

TensorFlow: biblioteca para computación numérica orientada a aprendizaje automático.

- núcleo similar a NumPy pero con soporte para GPU
- soporte para computación distribuída (e.g. clusters)
- compilador que optimiza para velocidad y uso de memoria
- extrae grafo computacional que se exporta a formato portable
- provee diferenciación automática (autodiff) y optimizadores



API de TensorFlow [†]

[†] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022

Tensores y operaciones

Tensor: arreglo multidimensional

Tensores y operaciones

Tensor: arreglo multidimensional

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.constant(42) # scalar

>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Tensores y operaciones

Tensor: arreglo multidimensional

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.constant(42) # scalar

>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32

>>> t1 = t[:, 1:]
>>> t1.shape
TensorShape([2, 2])
>>> t2 = t[..., 1, tf.newaxis]
>>> t2.shape
TensorShape([2, 1])
```

Tensores y operaciones

Tensor: arreglo multidimensional

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.constant(42) # scalar

>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32


>>> t1 = t[:, 1:]
>>> t1.shape
TensorShape([2, 2])
>>> t2 = t[..., 1, tf.newaxis]
>>> t2.shape
TensorShape([2, 1])
```

```
>>> t + 10
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>


>>> tf.square(t)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>


>>> t @ tf.transpose(t)
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Tensores y operaciones

Tensor: arreglo multidimensional

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.constant(42) # scalar

>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32


>>> t1 = t[:, 1:]
>>> t1.shape
TensorShape([2, 2])
>>> t2 = t[..., 1, tf.newaxis]
>>> t2.shape
TensorShape([2, 1])

>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
# en la práctica: add_weight()
```

```
>>> t + 10
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>


>>> tf.square(t)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>


>>> t @ tf.transpose(t)
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

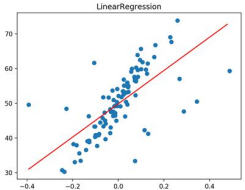
Tensores y NumPy

```
>>> a = np.array([2., 4., 5.])

>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])

>>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
[4., 5., 6.]], dtype=float32)

>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>

>>> np.square(t)
array([[ 1., 4., 9.],
[16., 25., 36.]], dtype=float32)
```

Función de costo personalizada

problema de regresión ruidoso (outliers)

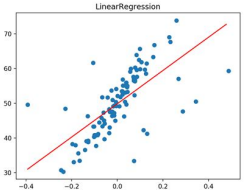
- MSE error cuadrático medio

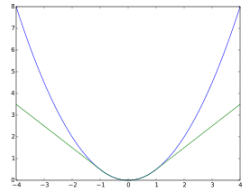


Función de costo personalizada

problema de regresión ruidoso (outliers)

- MSE error cuadrático medio
- MAE error absoluto medio



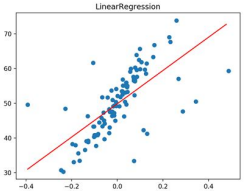


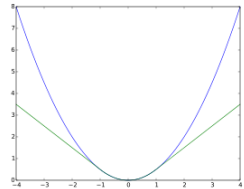
Función de costo personalizada

problema de regresión ruidoso (outliers)

- MSE error cuadrático medio
- MAE error absoluto medio
- Huber loss:

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & |e| < \delta \\ \delta|e| - \frac{1}{2}\delta^2 & |e| \geq \delta \end{cases}$$



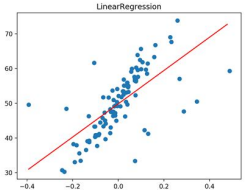


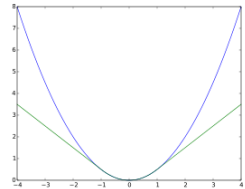
Función de costo personalizada

problema de regresión ruidoso (outliers)

- MSE error cuadrático medio
- MAE error absoluto medio
- Huber loss:

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & |e| < \delta \\ \delta|e| - \frac{1}{2}\delta^2 & |e| \geq \delta \end{cases}$$






```
def create_huber(threshold=1.0):
        def huber_fn(y_true, y_pred):
                error = y_true - y_pred
                is_small_error = tf.abs(error) < threshold
                squared_loss = tf.square(error) / 2
                linear_loss = threshold * tf.abs(error) - threshold**2 / 2
                return tf.where(is_small_error, squared_loss, linear_loss)
        return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Función de costo personalizada

Guardar y recuperar el modelo.

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

Función de costo personalizada

Guardar y recuperar el modelo. ATENCIÓN: No se guarda el valor del umbral.

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

Función de costo personalizada

Guardar y recuperar el modelo. ATENCIÓN: No se guarda el valor del umbral.

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})

class HuberLoss(keras.losses.Loss):
        def __init__(self, threshold=1.0, **kwargs):
                self.threshold = threshold
                super().__init__(**kwargs)
        def call(self, y_true, y_pred):
                error = y_true - y_pred
                is_small_error = tf.abs(error) < self.threshold
                squared_loss = tf.square(error) / 2
                linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
                return tf.where(is_small_error, squared_loss, linear_loss)
        def get_config(self):
                base_config = super().get_config()
                return {**base_config, "threshold": self.threshold}

model.compile(loss=HuberLoss(2.), optimizer="nadam")

model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

Medida de desempeño personalizada

En la mayoría de los casos es igual a definir una función de costo personalizada.

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Para cada batch se calcula la métrica y se mantiene su valor medio desde el inicio de la época.

Medida de desempeño personalizada

En la mayoría de los casos es igual a definir una función de costo personalizada.

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Para cada batch se calcula la métrica y se mantiene su valor medio desde el inicio de la época.

Pero a veces se requiere una métrica que se actualice en cada batch (streaming metric).

Medida de desempeño personalizada

En la mayoría de los casos es igual a definir una función de costo personalizada.

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Para cada batch se calcula la métrica y se mantiene su valor medio desde el inicio de la época.

Pero a veces se requiere una métrica que se actualice en cada batch (streaming metric).

Precision en clasificación binaria

- batch 1: y = [0, 1, 1, 1, 0, 1, 0, 1], y_pred = [1, 1, 0, 1, 0, 1, 0, 1]
- batch 2: y = [0, 1, 0, 0, 1, 0, 1, 1], y_pred = [1, 0, 1, 1, 0, 0, 0, 0]

Medida de desempeño personalizada

En la mayoría de los casos es igual a definir una función de costo personalizada.

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Para cada batch se calcula la métrica y se mantiene su valor medio desde el inicio de la época.

Pero a veces se requiere una métrica que se actualice en cada batch (streaming metric).

Precision en clasificación binaria

- batch 1: y = [0, 1, 1, 1, 0, 1, 0, 1], y_pred = [1, 1, 0, 1, 0, 1, 0, 1] Precision = 80%
- batch 2: y = [0, 1, 0, 0, 1, 0, 1, 1], y_pred = [1, 0, 1, 1, 0, 0, 0, 0] Precision = 0%

Medida de desempeño personalizada

En la mayoría de los casos es igual a definir una función de costo personalizada.

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Para cada batch se calcula la métrica y se mantiene su valor medio desde el inicio de la época.

Pero a veces se requiere una métrica que se actualice en cada batch (streaming metric).

Precision en clasificación binaria

- batch 1: y = [0, 1, 1, 1, 0, 1, 0, 1], y_pred = [1, 1, 0, 1, 0, 1, 0, 1] Precision = 80%
- batch 2: y = [0, 1, 0, 0, 1, 0, 1, 1], y_pred = [1, 0, 1, 1, 0, 0, 0, 0] Precision = 0%

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

Medida de desempeño personalizada

```python
class HuberMetric(keras.metrics.Metric):
        def __init__(self, threshold=1.0, **kwargs):
                super().__init__(**kwargs) # handles base args (e.g., dtype)
                self.threshold = threshold
                self.huber_fn = create_huber(threshold)
                self.total = self.add_weight("total", initializer="zeros")
                self.count = self.add_weight("count", initializer="zeros")
        def update_state(self, y_true, y_pred, sample_weight=None):
                metric = self.huber_fn(y_true, y_pred)
                self.total.assign_add(tf.reduce_sum(metric))
                self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
        def result(self):
                return self.total / self.count
        def get_config(self):
                base_config = super().get_config()
                return {**base_config, "threshold": self.threshold}
```

Cálculo de gradientes con autodiff

```
def f(w1, w2):
        return 3 * w1 ** 2 + 2 * w1 * w2
```

Cálculo de gradientes con autodiff

```
def f(w1, w2):
        return 3 * w1 ** 2 + 2 * w1 * w2
```

Aproximación por diferencia finita.

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Cálculo de gradientes con autodiff

```
def f(w1, w2):
        return 3 * w1 ** 2 + 2 * w1 * w2
```

Usando autodiff

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
        z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Aproximación por diferencia finita.

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Entrenamiento personalizado

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
        keras.layers.Dense(30, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=l2_reg),
        keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Entrenamiento personalizado

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
        keras.layers.Dense(30, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=l2_reg),
        keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

```
def random_batch(X, y, batch_size=32):
        idx = np.random.randint(len(X), size=batch_size)
        return X[idx], y[idx]
```

Entrenamiento personalizado

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
        keras.layers.Dense(30, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=l2_reg),
        keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

```
def random_batch(X, y, batch_size=32):
        idx = np.random.randint(len(X), size=batch_size)
        return X[idx], y[idx]
```

```
def print_status_bar(iteration, total, loss, metrics=None):
        metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
        for m in [loss] + (metrics or [])])
                end = "" if iteration < total else "\n"
                print("\r{}/{} - ".format(iteration, total) + metrics, end=end)
```

Entrenamiento personalizado

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
        keras.layers.Dense(30, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=l2_reg),
        keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

```
def random_batch(X, y, batch_size=32):
        idx = np.random.randint(len(X), size=batch_size)
        return X[idx], y[idx]
```

```
def print_status_bar(iteration, total, loss, metrics=None):
        metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
        for m in [loss] + (metrics or [])])
                end = "" if iteration < total else "\n"
                print("\r{}/{} - ".format(iteration, total) + metrics, end=end)
```

Entrenamiento personalizado

```
for epoch in range(1, n_epochs + 1):
        print("Epoch {}/{}".format(epoch, n_epochs))
        for step in range(1, n_steps + 1):
                X_batch, y_batch = random_batch(X_train_scaled, y_train)
                with tf.GradientTape() as tape:
                        y_pred = model(X_batch, training=True)
                        main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
                        loss = tf.add_n([main_loss] + model.losses)
                gradients = tape.gradient(loss, model.trainable_variables)
                optimizer.apply_gradients(zip(gradients, model.trainable_variables))
                mean_loss(loss)
                for metric in metrics:
                        metric(y_batch, y_pred)
                print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
        print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
        for metric in [mean_loss] + metrics:
                metric.reset_states()
```

Motivación

- Los datos no siempre se pueden almacenar en un arreglo *numpy* porque no entran en memoria
- Además:
  - Los datos pueden provenir de múltiples fuentes
  - Es necesario (texto) o se quiere (imágenes) realizar un preprocesamiento para generar las características
- Hacer una lectura eficiente de los datos, implica manejar: *multithreading*, *queuing*, *batching* y *prefetching*.

Data API

- La Data API basa su funcionamiento entorno a una abstracción llamada *Dataset*
- Un *Dataset* es una secuencia de items que se genera a partir de una fuente (lista de archivos, pandas data frames, Tensorflow Records)
- Ej: a partir de un *tensor*

```
>>> X = tf.range(10) # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

Data API

- La Data API basa su funcionamiento entorno a una abstracción llamada *Dataset*
- Un *Dataset* es una secuencia de items que se genera a partir de una fuente (lista de archivos, pandas data frames, Tensorflow Records)
- Ej: a partir de un *tensor*

```
>>> X = tf.range(10) # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

Transformación del Dataset

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

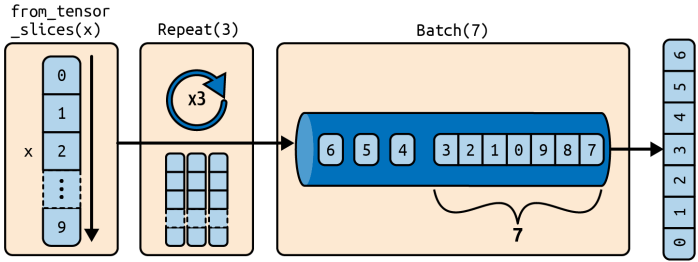


Figure: Transformaciones encadenadas en un dataset

Shuffling

- ¿Por qué es importante y no trivial desordenar los datos?

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...
print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

Entrelazado

- Cuando se lee de múltiples fuentes se entrelazan los datos
- A partir de una lista de fuentes

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]

filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Entrelazado

- Cuando se lee de múltiples fuentes se entrelazan los datos
- A partir de una lista de fuentes

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]

filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

- Se genera un dataset

```
n_readers = 5
dataset = filepath_dataset.interleave(
lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
cycle_length=n_readers)
```

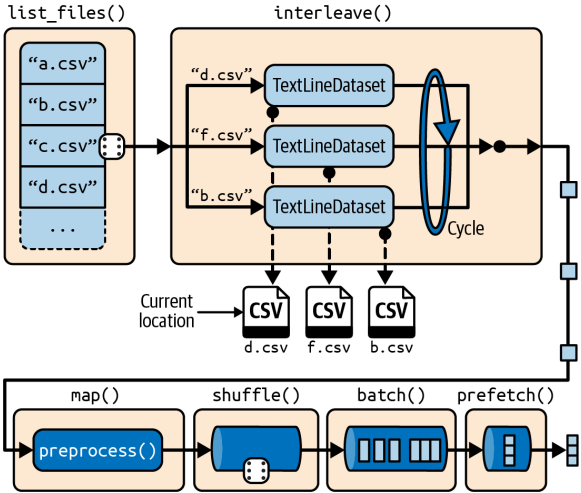
Transformación del dataset

- Funciones para transformar el dataset

```
>>> dataset = dataset.map(lambda x: x * 2)

>>> dataset = dataset.apply(tf.data.experimental.unbatch())

>>> dataset = dataset.filter(lambda x: x < 10)
```

Ejemplo

- Carga y preprocesamiento de archivos de texto



Ejemplo

- Carga y preprocesamiento de archivos de texto

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
            cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    return dataset.batch(batch_size).prefetch(1)
```

Prefetching

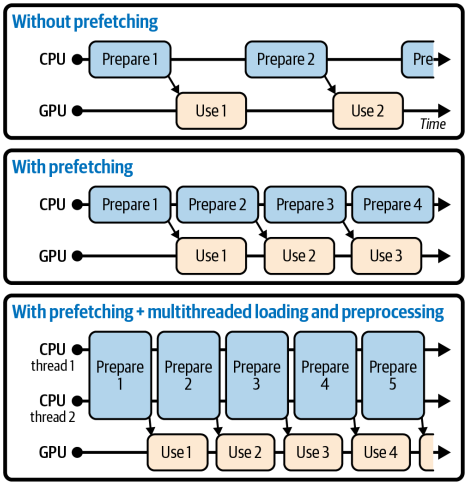


Figure: Prefetching

Entrenamiento con el dataset

```
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Ejemplos de tf.data: Build TensorFlow input pipelines

- Generación de dataset a partir de texto. En sección: consuming text files.
- Generación de dataset a partir de caminos a imágenes
  - Levantar las imágenes. En sección: consuming sets of files.
  - Generar el dataset. En sección: decoding image data.

TF Records

- Formato preferido para almacenar gran cantidad de datos
- Formato binario que contiene: datos, largo de datos y dos CRC checksums para detectar errores.
- Escritura

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

- Lectura

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

Tensorflow Protobufs

- En el ejemplo anterior se guardaron *strings* en un TF Record
- Tensorflow prefiere guardar la información en Protobufs (*Protocol buffer*) serializados
- Una instancia de un dataset de almacena en un *protobuf* llamado *Example*.

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
oneof kind {
BytesList bytes_list = 1;
FloatList float_list = 2;
Int64List int64_list = 3;
}
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

Ejemplo - Escritura de TF Record

- Se guarda en un *TF Record* una instancia de un dataset con los atributos:
  - Name: Alice
  - id: 123
  - emails: a@b.com c@d.com

Ejemplo - Escritura de TF Record

- Se guarda en un *TF Record* una instancia de un dataset con los atributos:
  - Name: Alice
  - id: 123
  - emails: a@b.com c@d.com

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example
person_example = Example(
    features=Features(
        feature={
        "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
         "id": Feature(int64_list=Int64List(value=[123])),
         "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                       b"c@d.com"]))
})

with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Ejemplo - Lectura de TF Record

- Primero se definen los elementos a leer

```
feature_description = {
"name": tf.io.FixedLenFeature([], tf.string, default_value=""),
"id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
"emails": tf.io.VarLenFeature(tf.string),
}
```

Ejemplo - Lectura de TF Record

- Primero se definen los elementos a leer

```
feature_description = {
"name": tf.io.FixedLenFeature([], tf.string, default_value=""),
"id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
"emails": tf.io.VarLenFeature(tf.string),
}
```

- Se levanta el dataset

```
dataset=tf.data.TFRecordDataset(["my_contacts.tfrecord"])
for serialized_example in dataset:
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

- Se acede a los elementos

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'],
[...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'],
[...])>
```

Observaciones

- Se recomienda la utilización de *TF Records* cuando el cuello de botella es la lectura de los datos
- Reglas de pulgar:
  - Repartir los datos de entrenamiento en al menos 10 archivos
  - El tamaño óptimo de cada archivo es del orden de cientos de MB.
- Ver ejemplos de lectura y escritura con TF Records

Referencias

A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Inc., 2019.