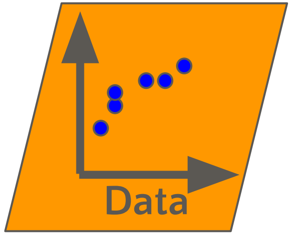
Loading and PreProcessing Data in TensorFlow

Project

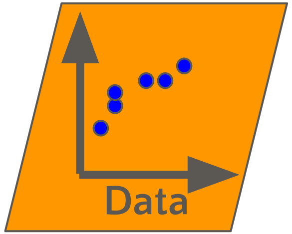
Motivation - Why this Chapter

Recap

Recap

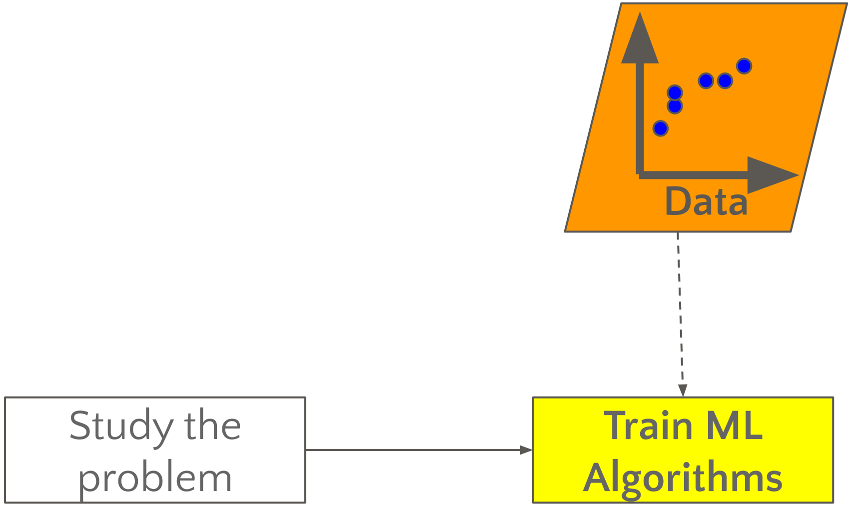


Recap

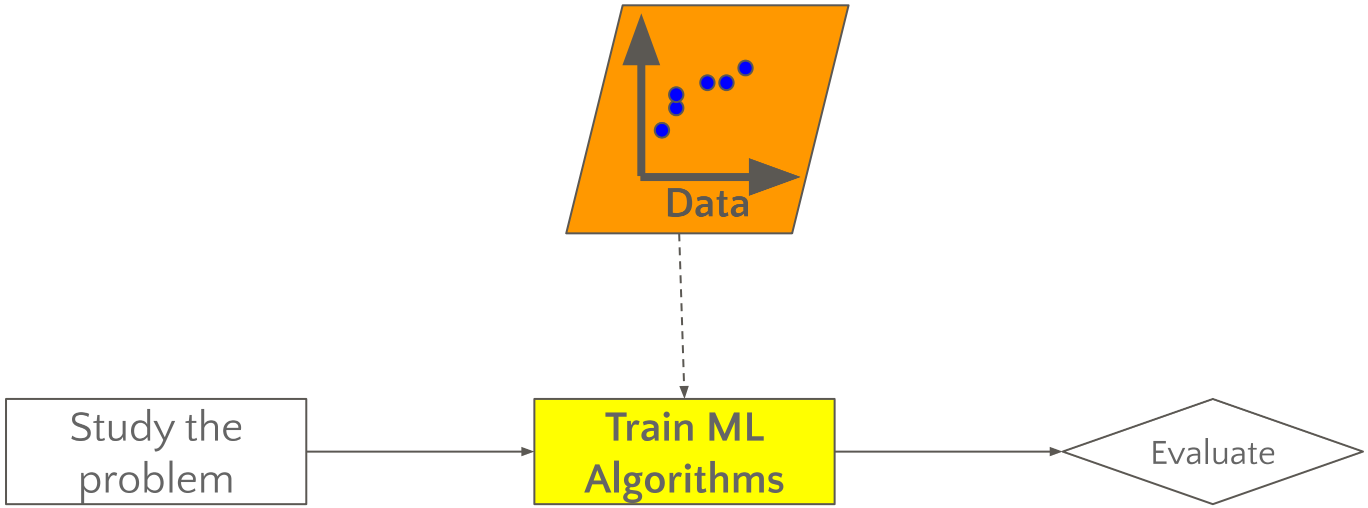


Study the problem

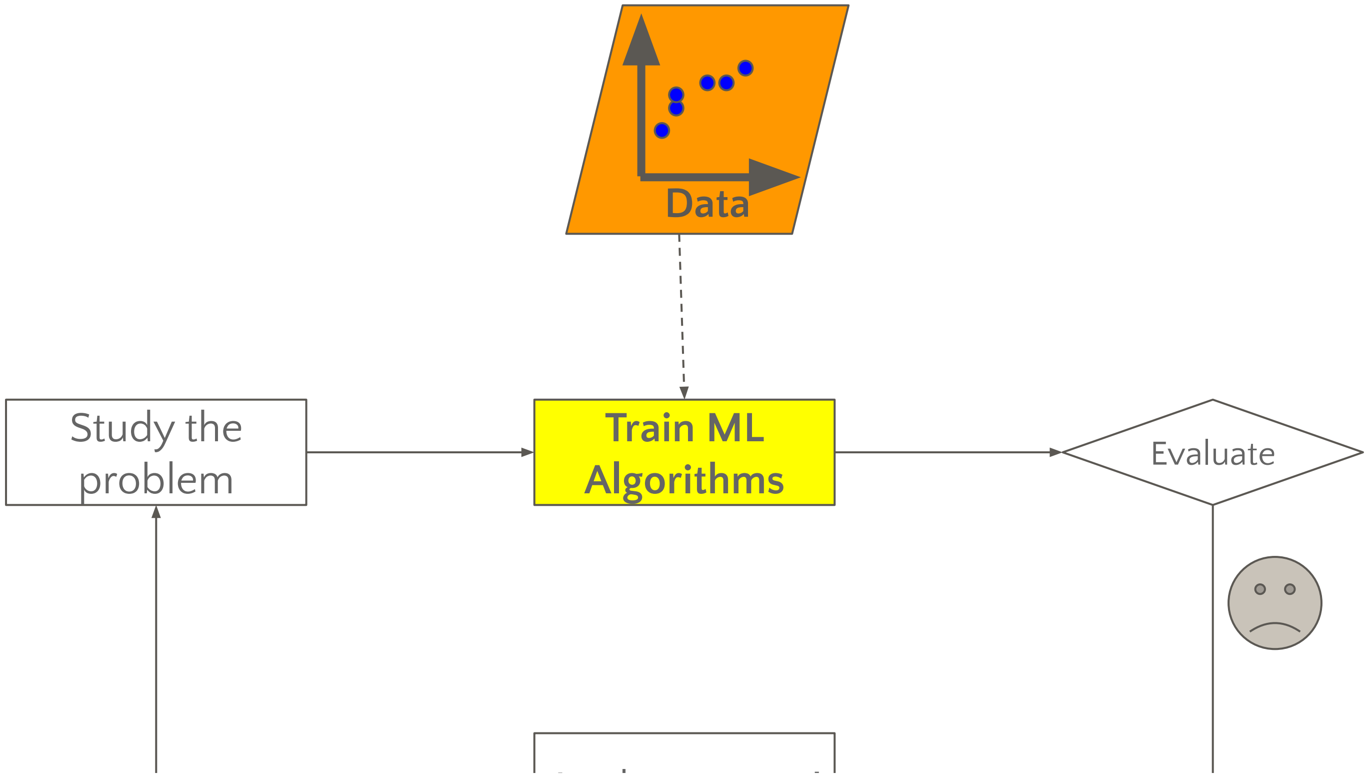
Recap



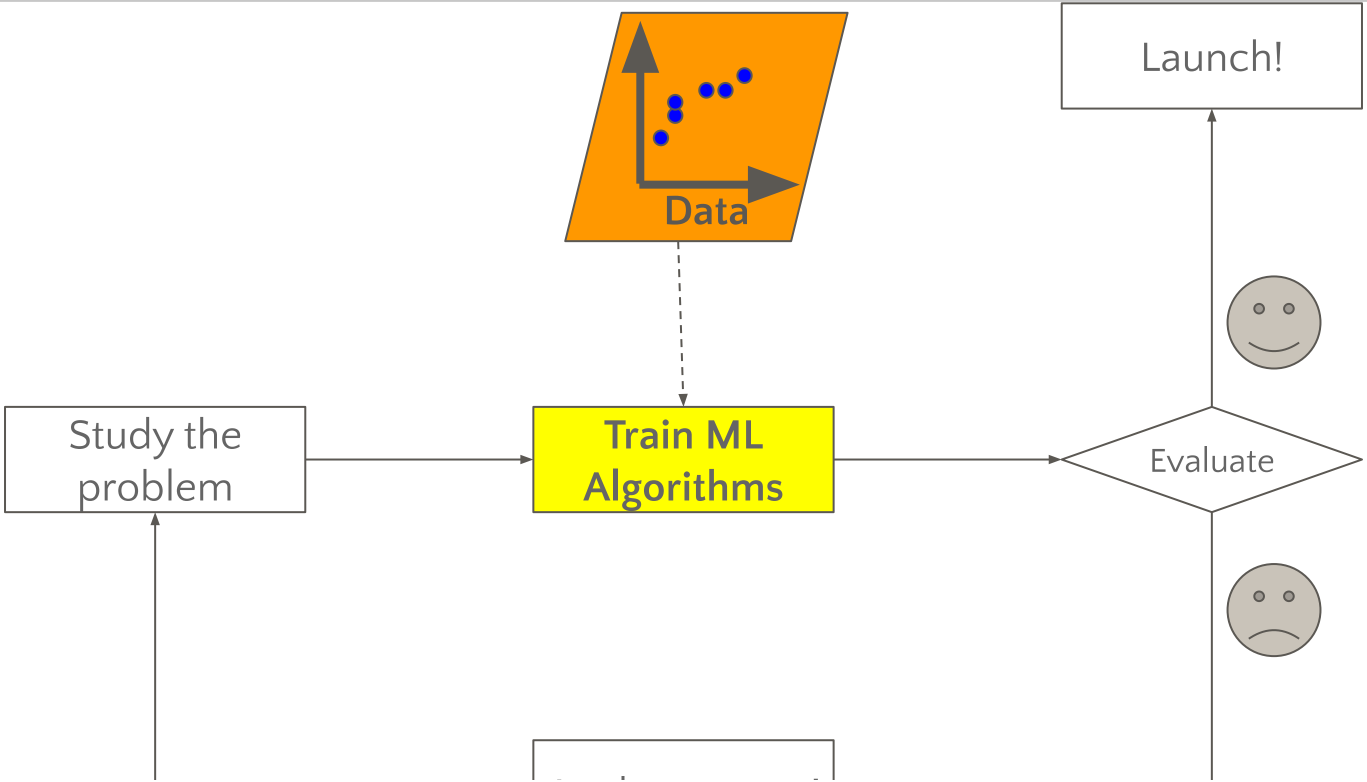
Recap



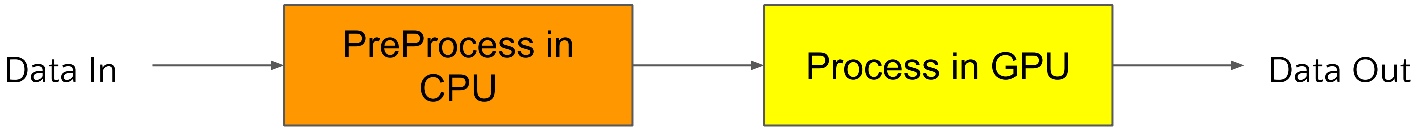
Recap



Recap


Data
Launch!
Study the problem
Train ML Algorithms
Evaluate

Data Processing inside the Machine



All Data is in the RAM

- If Data size is small this is fine

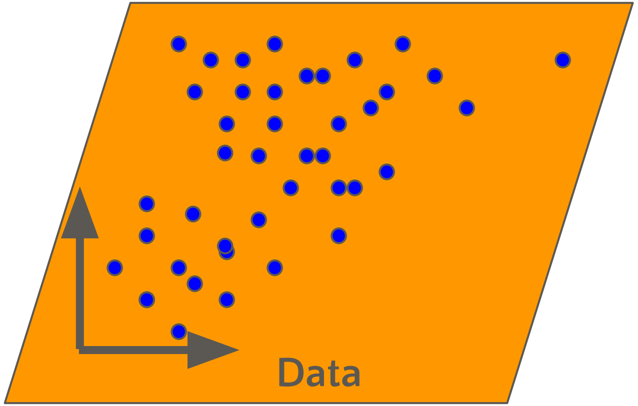
All Data is in the RAM

- If Data size is small this is fine

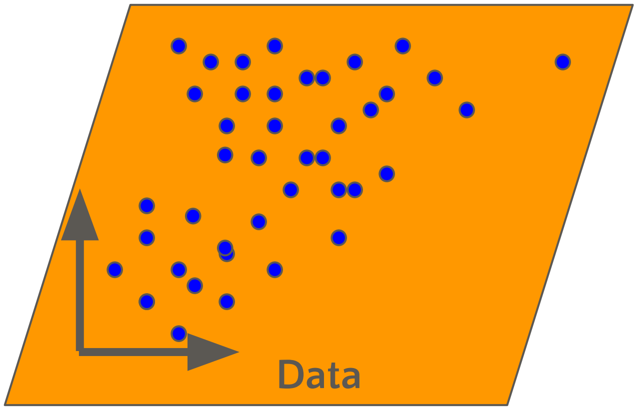
What happens if the data is more than the RAM?

For e.g. ImageNet has 14 million images > 150 GB

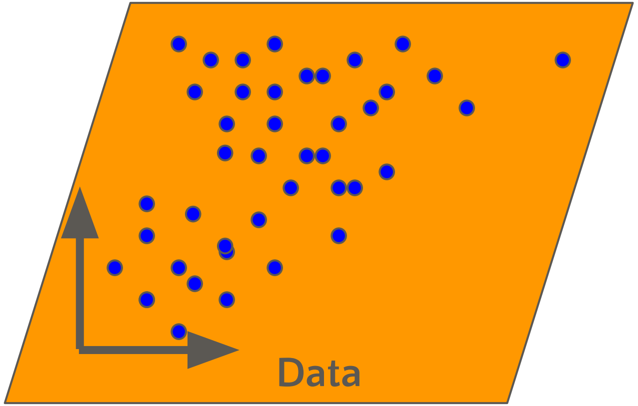
How to Manage Big Data



Break it up to Small Data

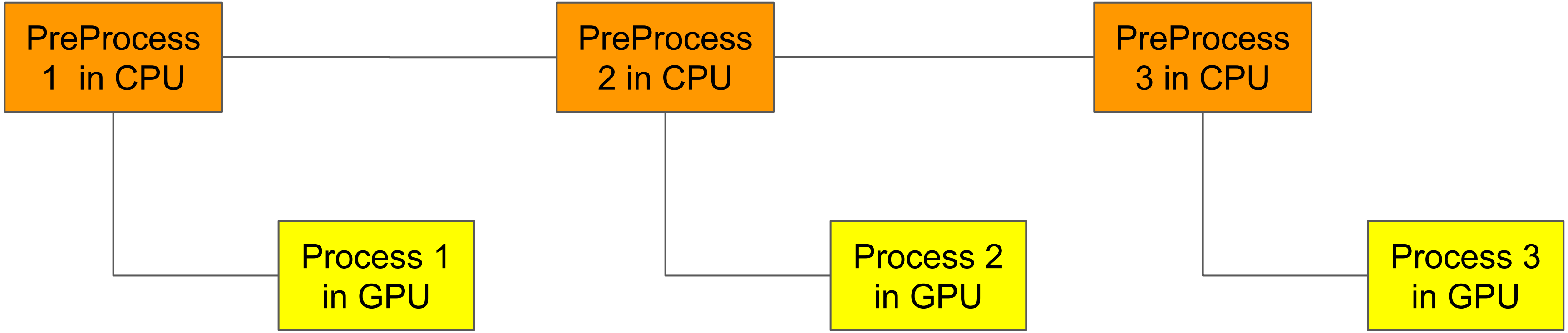


Break it up to Small Data

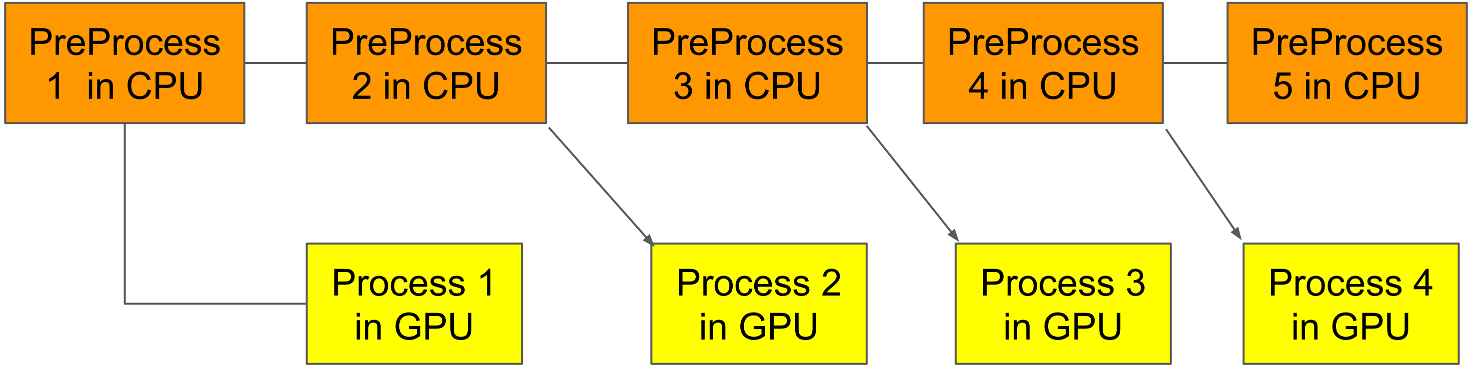


BATCHING

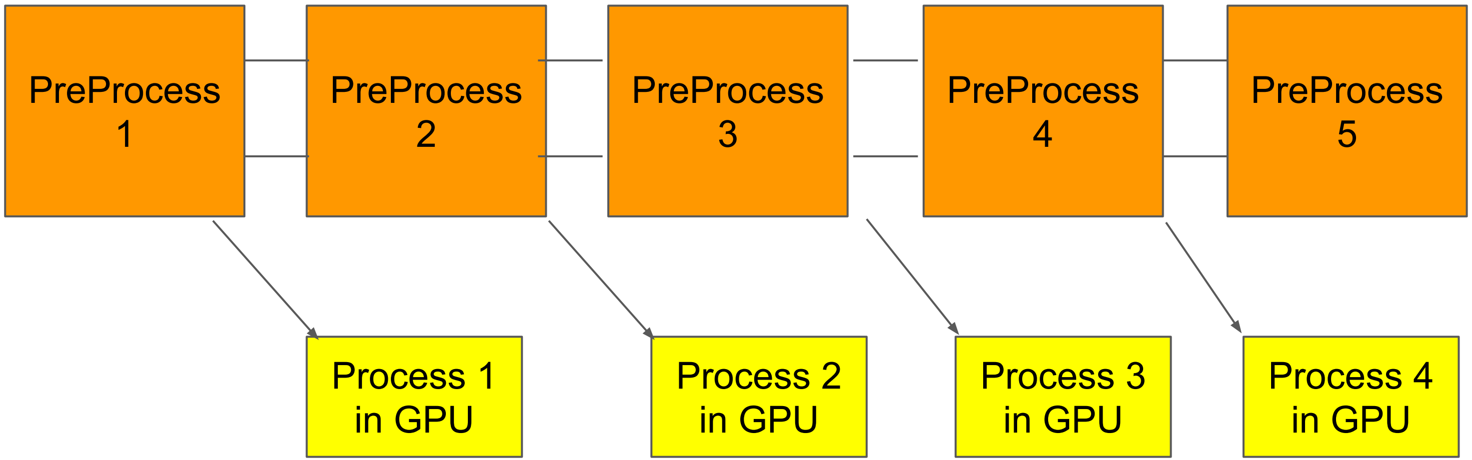
Processing Batches



PreFetch


PreProcess 1 in CPU
PreProcess 2 in CPU
PreProcess 3 in CPU
PreProcess 4 in CPU
PreProcess 5 in CPU
Process 1 in GPU
Process 2 in GPU
Process 3 in GPU
Process 4 in GPU

Multithreading



Tensorflow Data API

- Batching
- Prefetching
- Pipeline
- Leverage multi-core capabilities
- Supports multithreaded operations

Multiple Input Types

- Text files like CSV
- Binary files
- TFRecord (TensorFlow specific binary format)
- SQL Databases
- Big Query

Transformation of Data

- Data often needs transformation during preprocessing
- One hot encoding
- Embedding
- Bag of Words encoding

Hands on with TF Data API

- We will first demonstrate with data from the RAM
- We will then demonstrate with data from multiple files

Hands on with TF Data API

```
X = tf.range(10)
dataset = tf.data.Dataset.from_tensor_slices(X)
```

Hands on with TF Data API

```
X = tf.range(10)
dataset = tf.data.Dataset.from_tensor_slices(X)
```

Created a dataset from Tensors

Dataset

```
for item in dataset:
    print(type(item), item)
```

```
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(0, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(1, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(2, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(3, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(4, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(5, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(6, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(7, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(8, shape=(), dtype=int32)
<class 'tensorflow.python.framework.ops.EagerTensor'> tf.Tensor(9, shape=(), dtype=int32)
```

Looks like a list of Tensors containing 1 integer each

Example of few transformations

```
dataset_3 = dataset.repeat(3)
```

Creates a new dataset repeating original dataset 3 times

Chaining a command to create Batches

```
dataset=dataset.repeat(3).batch(7)
for item in dataset_batched:
    print(item)
```

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int64)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int64)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int64)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int64)
tf.Tensor([8 9], shape=(2,), dtype=int64)
```

Chaining a command to create Batches

```
dataset=dataset.repeat(3).batch(7)
for item in dataset_batched:
    print(item)
```

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int64)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int64)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int64)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int64)
tf.Tensor([8 9], shape=(2,), dtype=int64)
```

The repeat and batch command can be run as a chain. Note that command does not modify the original dataset. It returns a new one.

Additional Transformation: Map

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

```
for item in dataset:
    print(item)
```

```
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int64)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int64)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int64)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int64)
tf.Tensor([16 18], shape=(2,), dtype=int64)
```

Using map and lambda, we transformed the entire dataset by doubling it

Unbatching

```
dataset = dataset.unbatch()
for item in dataset:
    print(item)
```

```
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
tf.Tensor(10, shape=(), dtype=int64)
tf.Tensor(12, shape=(), dtype=int64)
tf.Tensor(14, shape=(), dtype=int64)
tf.Tensor(16, shape=(), dtype=int64)
tf.Tensor(18, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
```

Returns a unbatched dataset

Filter

```python
dataset = dataset.filter(lambda x: x < 10)  # keep only items < 10
for item in dataset:
    print(item)
```

```
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
tf.Tensor(6, shape=(), dtype=int64)
tf.Tensor(8, shape=(), dtype=int64)
```

Shuffling

Shuffling

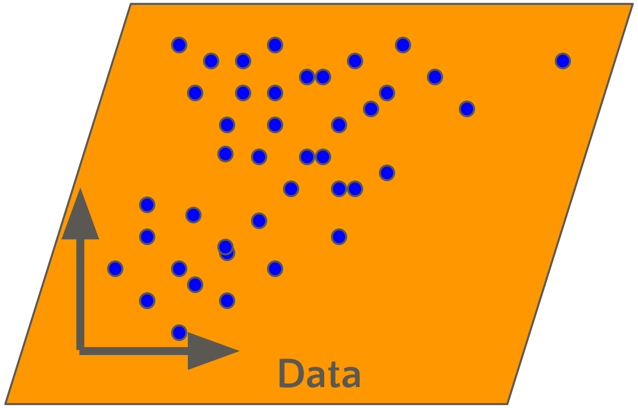
```python
tf.random.set_seed(42)

dataset = tf.data.Dataset.range(10).repeat(3)
dataset = dataset.shuffle(buffer_size=3, seed=42).batch(7)
for item in dataset:
    print(item)
```

```
tf.Tensor([1 3 0 4 2 5 6], shape=(7,), dtype=int64)
tf.Tensor([8 7 1 0 3 2 5], shape=(7,), dtype=int64)
tf.Tensor([4 6 9 8 9 7 0], shape=(7,), dtype=int64)
tf.Tensor([3 1 4 5 2 8 7], shape=(7,), dtype=int64)
tf.Tensor([6 9], shape=(2,), dtype=int64)
```

Shuffling with Tensorflow Data API. Note the numbers have a random sequence as compared to before

How to work with Big Data

Small files can be read into the RAM



California Dataset

- The California Dataset is broken into multiple files and provided separately as train, valid and test
- The mean and median of the input features are available

Train_dataset sample: datasets/housing/my_train_00.csv (20 files)
Valid_dataset sample: datasets/housing/my_valid_00.csv (10 files)
Test_dataset sample: datasets/housing/my_test_00.csv (10 files)

Data Inspection

```python
import pandas as pd

pd.read_csv(train_filepaths[0]).head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedianHouseValue
0	3.5214	15.0	3.049945	1.106548	1447.0	1.605993	37.63	-122.43	1.442
1	5.3275	5.0	6.490060	0.991054	3464.0	3.443340	33.69	-117.39	1.687
2	3.1000	29.0	7.542373	1.591525	1328.0	2.250847	38.44	-122.98	1.621
3	7.1736	12.0	6.289003	0.997442	1054.0	2.695652	33.55	-117.70	2.621
4	2.0549	13.0	5.312457	1.085092	3297.0	2.244384	33.93	-116.93	0.956

We will not use pandas in further. This just for visualising the data

Shuffling and Interleaving

Shuffling and Interleaving

We will do this for data from multiple files, by interleaving lines from different files together. Like shuffling cards from multiple decks.

Shuffling and Interleaving with TF Data

- Step 1

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

Make a dataset with a list of files. By default the list is shuffled. This is like shuffling your decks

Shuffling and Interleaving with TF Data

- Step 2

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

Shuffling and Interleaving with TF Data

- Step 2

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

Read 5 files at a time

Shuffling and Interleaving with TF Data

- Step 2

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

Read 5 files at a time
Remove the 1st line which contains headers

Shuffling and Interleaving with TF Data

- Step 2

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

Read 5 files at a time
Remove the 1st line which contains headers
Interleave

PreProcessing the Data

- By default the data loaded will be strings

```
b'4.5909,16.0,5.475877192982456,1.0964912280701755,1357.0,2.9758771929824563,33.63,-117.71,2.418'
b'2.4792,24.0,3.4547038327526134,1.1341463414634145,2251.0,3.921602787456446,34.18,-118.38,2.0'
b'4.2708,45.0,5.121387283236994,0.953757225433526,492.0,2.8439306358381504,37.48,-122.19,2.67'
b'2.1856,41.0,3.7189873417721517,1.0658227848101265,803.0,2.0329113924050635,32.76,-117.12,1.205'
b'4.1812,52.0,5.701388888888889,0.9965277777777778,692.0,2.4027777777777777,33.73,-118.31,3.215'
```

- The data needs to be pre-processed

PreProcessing with tf.io.decode_csv

- tf.io.decode_csv is a function for processing csv strings
- It takes two inputs, the string to be processed and array containing default values, number and types of columns
- It returns a list of scalar tensors

PreProcessing function

```
n_inputs = 8 # X_train.shape[-1]
def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

PreProcessing function

```
n_inputs = 8 # X_train.shape[-1]
def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

- The csv line inputs are floats and 0 by default
- n_inputs is the number of inputs
- The last value in def is an empty array of floats with no defaults, so an error is generated if it is missing

PreProcessing function

```
n_inputs = 8 # X_train.shape[-1]
def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

- Separate inputs features and labels
- The decode_csv function returns a list of scalar tensors (with one in each column), which is converted to 1D array

PreProcessing function

```
n_inputs = 8 # X_train.shape[-1]
def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

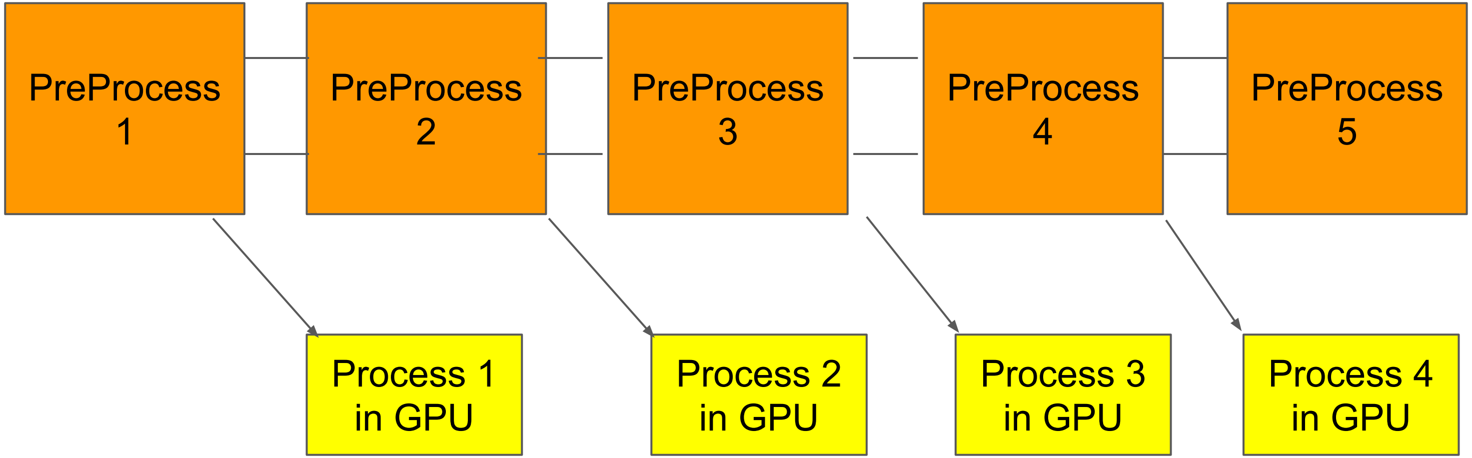
- Normalizing the inputs with the pre-calculated mean and standard deviation
- Returns normalized input vector and label

All put together

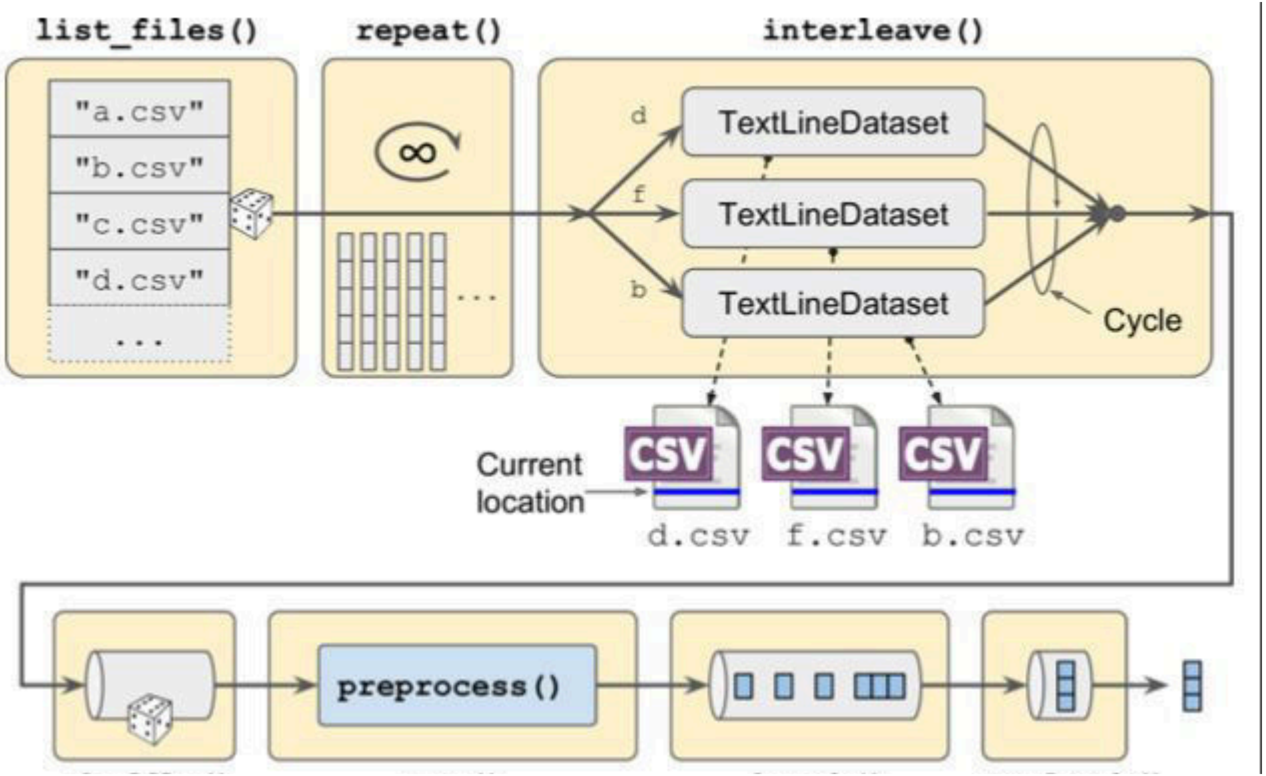
```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
    dataset = dataset.interleave(
        lambda 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)
    dataset = dataset.batch(batch_size)
    return dataset.prefetch(1)
```

Recall

Prefetch 1 enables one batch to be ready



Recall



list_files()
repeat()
interleave()
"a.csv"
"b.csv"
"c.csv"
"d.csv"
...
d
f
b
TextLineDataset
TextLineDataset
TextLineDataset
Cycle
Current location
CSV
CSV
CSV
d.csv
f.csv
b.csv
preprocess()

Reading batches

```
tf.random.set_seed(42)

train_set = csv_reader_dataset(train_filepaths, batch_size=3)
for X_batch, y_batch in train_set.take(2):
    print("X =", X_batch)
    print("y =", y_batch)
    print()
```

- The code above takes 2 sets of 3 batches
- Check results in the notebook

Datasets in Action

- Datasets can be used with keras
- Datasets can be created for train, valid and test sets
- Instead of send X_train etc. the Datasets can be sent to the fit and evaluate methods
- The Data API can also be used to create custom training loops. The Notebook has examples

TFRecords

What is TF Records

TFRecords

- It is TensorFlow’s preferred format for storing large amounts of data
- CSV files are not efficient to save images and audio
- TFRecords is a binary format
- Binary formats are easier for computer programs to manage especially where data is stored partially on the hard disk and loaded continuously
- The records can have different lengths
- CRC checksum to check the integrity of data

TFRecord

- Can be created with tf.io.TFRecordWriter
- Can be read with tf.data.TFRecordDataset
- tf.data.TFRecordDataset supports parallel reads
- TF Records also supports compressed files. We can write and read compressed files

TFRecord Demo

TFRecord Demo with the Notebook

Protocol Buffers

- Protobuf is a serialized protocol buffer
- TFRecords usually (not exclusively) use them
- Developed by Google and has been open sourced since 2008
- It needs protoc, a protobuf compiler, to generate access classes for various classes
- Has python APIs to create and read.

Protocol Buffers Example

```
%%writefile person.proto
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

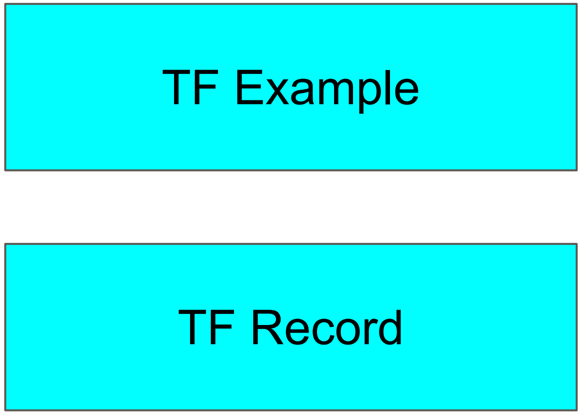
- Sample proto
- It needs to be compiled to its binary format that enables enables efficient serialisation
- This can then be read by TF

TensorFlow ProtoBufs

- TFRecords are binary strings which are mostly written with TF.Example

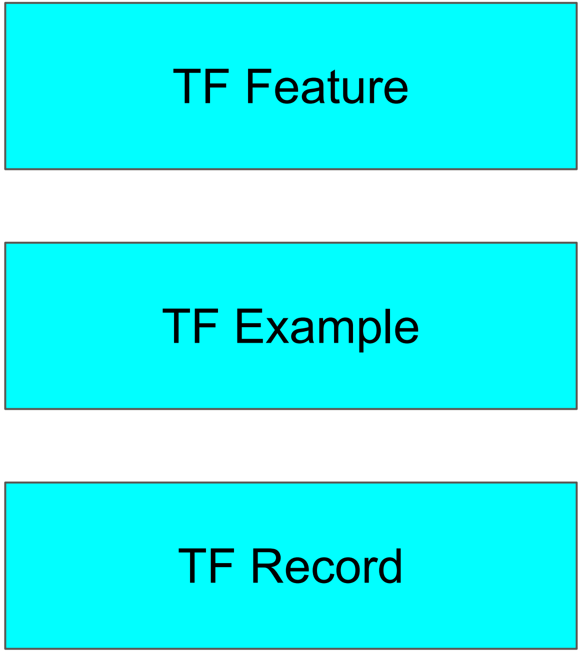
TF Record

TensorFlow ProtoBufs



- TFRecords are binary strings which are mostly written with TF.Example
- TF.Example is used as parent class

TensorFlow ProtoBufs



- TFRecords are binary strings which are mostly written with TF.Example
- TF.Example is used as parent class
- TF.Example uses TF.Feature

TF Example Proto

```
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; };
```

TF Example Proto

```
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; };
```

- BytesList can be used to save
  - Bytes
  - Strings

TF Example Proto

```
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; };
```

- FloatList can contain
  - Float (float32)
  - Double (float64)

TF Example Proto

```
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; };
```

- Int64List
  - Bool
  - Enum
  - Int32
  - Uint32
  - Int64
  - Uint64

TF Example Proto

```
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; };
```

- Int64List can contain
  - Bool
  - Enum
  - Int32
  - Uint32
  - Int64
  - Uint64

TF Example Proto

```
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; };
```

- Feature can contain
  - BytesList
  - FloatList
  - Int64List

TF Example Proto

```
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; };
```

- Features is a dictionary can contain a map of a feature name to a feature value

TF Example Proto

```
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; };
```

- Example contains a Features object, currently unused. Likely to be used in the future

Declaration of an 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"]))
        }))
```

- Declared Example with a Features dictionary
- Dictionary has names and values and Feature of different types

Writing a tf Record

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

- The data is serialised and converted to a string
- It is written to a file with TFRecordWriter

Reading a tf Record

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

- The serialised TF protobuf is read by parse_single_example
- It needs a string scalar tensor containing serialised data

Reading a tf Record

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

- The Feature description is a dictionary which maps name to each feature. Feature may be of 2 type
  - tf.io.FixedLenFeature
  - tf.io.VarLenFeature (emails here may have different lengths)
- The type of feature and default value can also be declared

Images with TFRecord

- TFRecords can also be used to store image data for processing
- tf.io.encode_jpeg() is used to encode an image and then convert it to a ByteList
- Serialised data from a TFRecord can be read and converted to an image using

References

1. https://chromium.googlesource.com/external/github.com/tensorflow/tensorflow/+/r0.10/tensorflow/g3doc/how_tos/tool_developers/index.md
2. https://www.tensorflow.org/tutorials/load_data/tfrecord
3. https://www.amazon.in/Hands-Machine-Learning-Scikit-Learn-TensorFlow/dp/1492032646/ref=sr_1_2?dchild=1&keywords=aurelien+geron&qid=1591935809&sr=8-2