Loading and Preprocessing Big Data

Data API, TFRecord, and TFDS

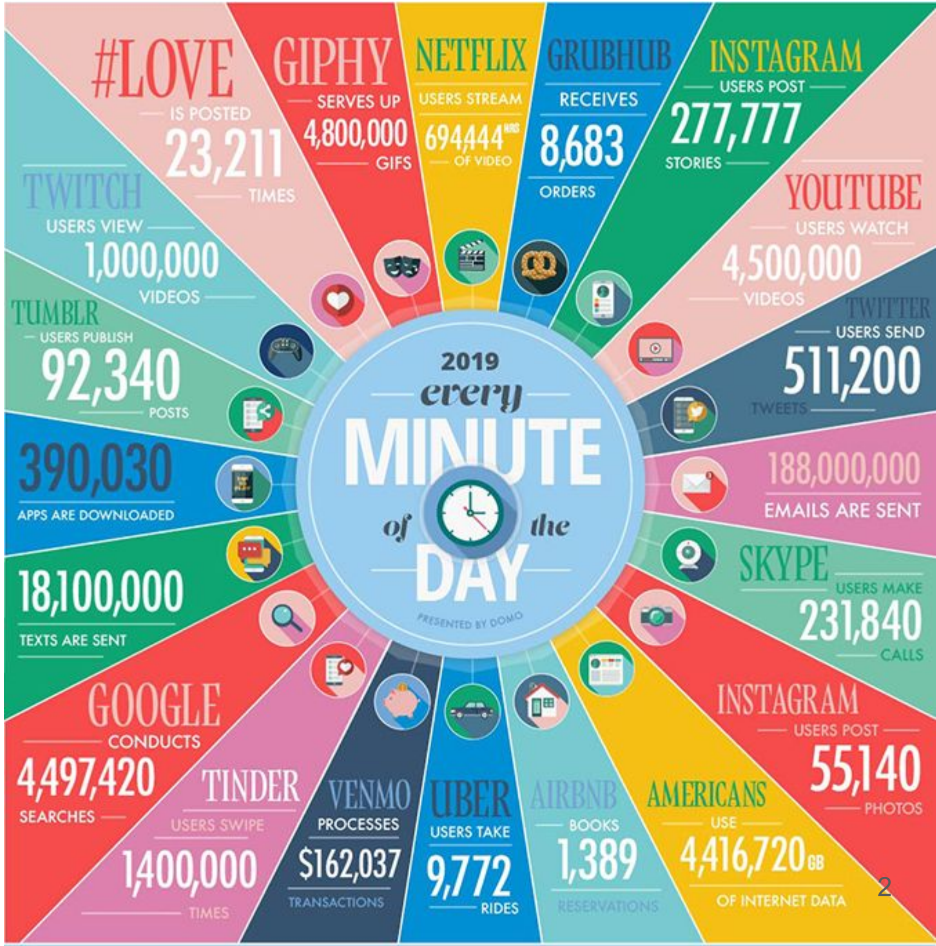
N. Rich Nguyen, PhD

SYS 6016

The age of Big Data

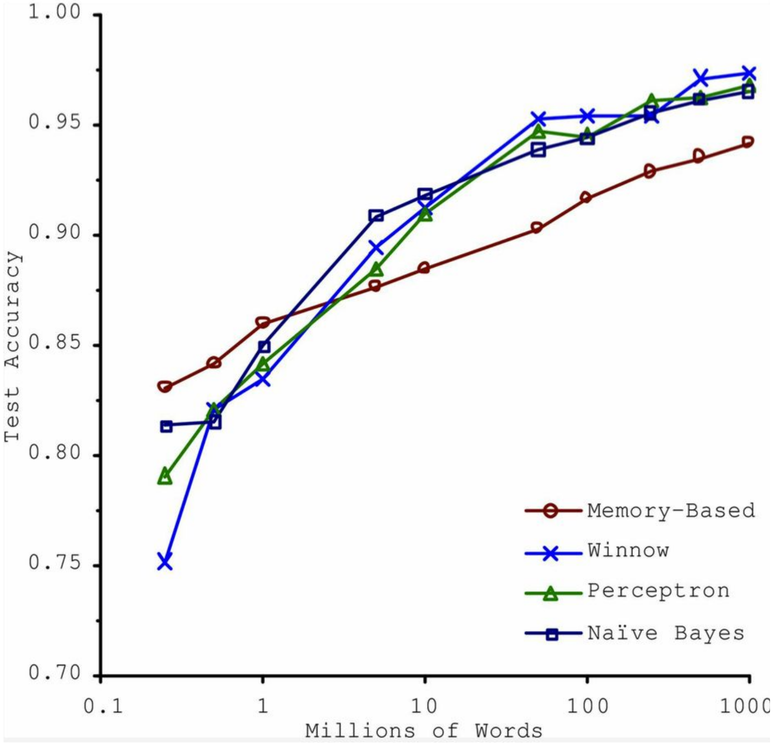
Source: Domo Infographics 2019

- Our lives are filled and surrounded with data of all kinds.
- Data is generated in ad clicks, likes on social media, shares, rides, transactions, streaming content, and so much more.
- When you put data in the hands of everyone, it can transform the way you think about the world.



Why doing ML on Big Data?

- The more training data ML models have, the better they get.
- *For example:* Let's look at a complex problem of natural language **disambiguation** (a project at Microsoft Research)
- "*I have _(to/too/two)_eggs for breakfast.*"
- The tradeoff between **data collection** and **algorithm development**



Best public datasets for ML

Dataset Finders: Google Dataset Search, Kaggle, UCI ML Repository, CMU Library, AWS Open Data, TensorFlow Datasets

- **General Datasets:** Boston Housing, Google Landmarks
- **Computer Vision Datasets:** xView, ImageNet, OpenImages
- **NLP Datasets:** IMDB Review, Amazon Reviews, SMS Spam Collection
- **Autonomous Driving Datasets:** Waymo, Berkeley DeepDrive, WPI
- **Clinical Datasets:** COVID-19, MIMIC-III

Challenges in Handling Big Datasets

Deep Learning systems are often trained on large datasets that will not fit in RAM (ie. **ImageNet** is 300GB, **OpenImages** is 561GB). **Great difficulty** in doing the following:

- Use **typical hardware** (ie. a workstation) to run data processing
- Preprocess and **normalize** the the data **at once**
- Truly **randomly select** a minibatch (for stochastic gradient descent)
- Assume data is **all numerical** (the world is not merely numerical, it is categorical, ordinal, textual and everything in between)

Data representations in neural networks

Data stored in multidimensional arrays, also called **tensors**.

A tensor is a container for **numerical** data (numbers): Scalars (0D tensors), Vectors (1D tensors), Matrices (2D tensors), Higher dim. arrays (3D+ tensors)

Deep learning models don't process an entire dataset at once, rather they break the data into small batches. When considering such a batch tensor, the **first axis** (axis 0), indexing the examples, is called the batch axis or **batch dimension**.

- Vector → 2D tensors of shape `(examples, features)`
- Timeseries/sequence → 3D tensors of shape `(examples, timesteps, features)`
- Images → 4D tensors of shape `(examples, height, width, channels)`
- Videos → 5D tensors of shape `(examples, frames, height, width, channels)`

The Data API and `Dataset` class

- The Data API revolves around the concept of a dataset which represents a sequence of data items. The Data API creates a **Dataset** object, tells it where to get data and how to transform it.
- TensorFlow takes care of all details (multithreading, queuing, and prefetching)
- For simplicity, let's *first* **create** a dataset entirely in RAM

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

```
<TensorSliceDataset shapes: (), types: tf.int32>
```

- **Iterating** over a dataset item is easy:

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

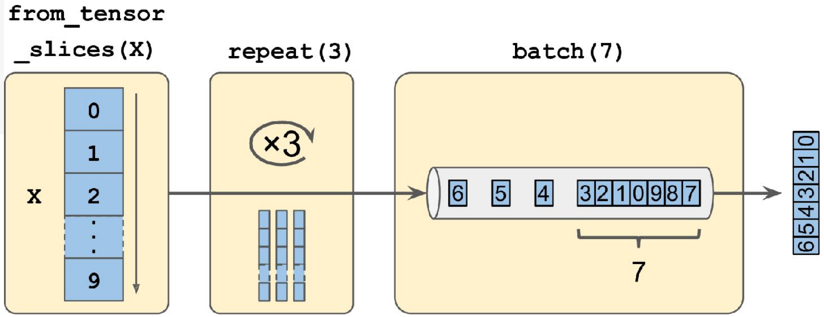
```
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(1, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(3, shape=(), dtype=int64)
```

Chaining Transformation

- Apply all sort of transformations by calling its **chain** transformations: we can **repeat**() the items on the dataset 3 times and group them in **batch**() of 7

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



- Transform using **map()** method

```
dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

- Break dataset into individual tensors using **unbatch()**

```
dataset = dataset.unbatch()
```

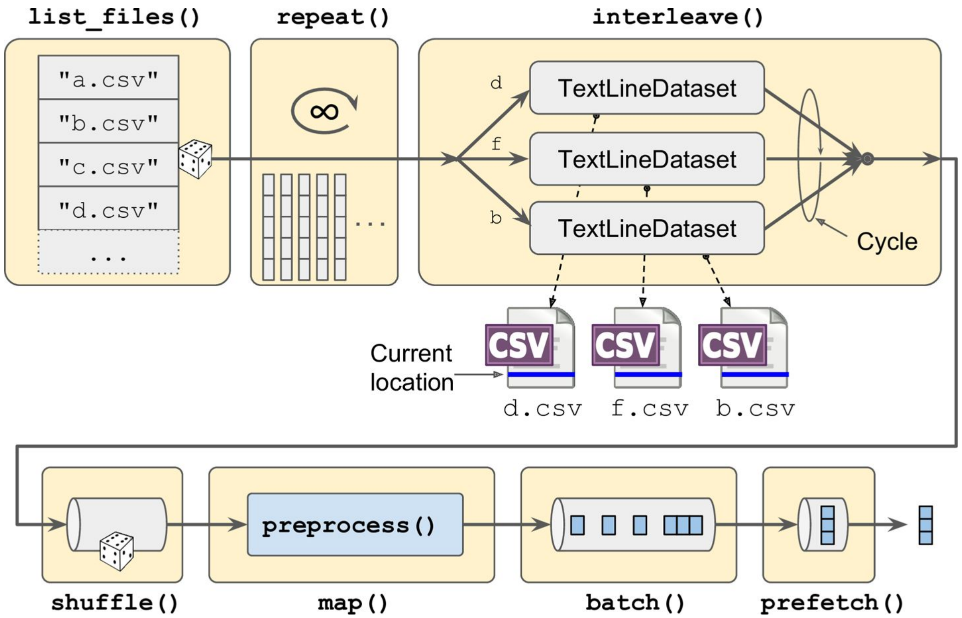
- Filter the dataset using **filter()**

```
dataset = dataset.filter(lambda x: x < 10)  # Items: [0,2,4,6,8,0,2,4,6]
```

Shuffling a Large Dataset

SGD works best when instances of training set are independent and identically distributed (i.i.d). We can ensure this property by shuffling the items using **shuffle**().

For large dataset (not fit in RAM), a solution is to shuffle the source data files itself: pick multiple files randomly, read them simultaneously, interleaving their records, and add a shuffling buffer. The Data API makes all this possible!



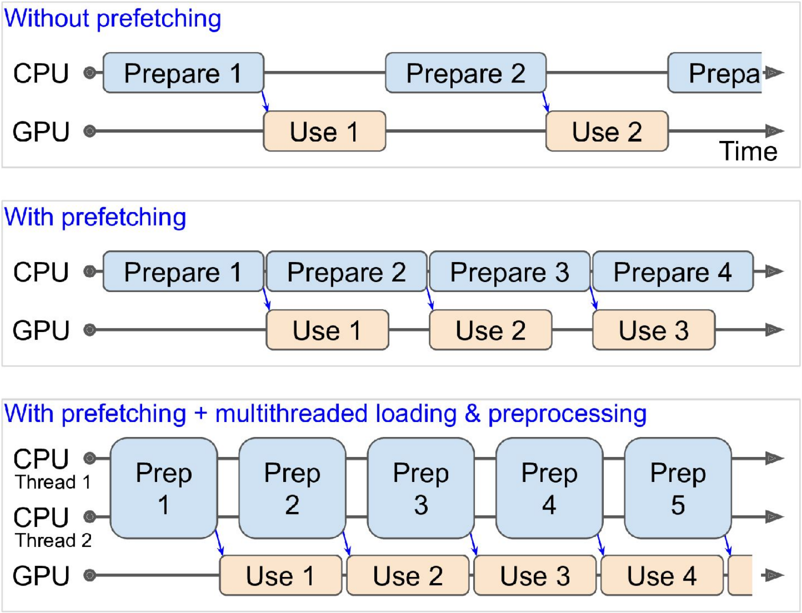
Loading data from multiple files

```
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)
```

- **list_file**() returns a dataset that shuffles the file paths
- **interleave**() creates a dataset that pulls multiple file paths, and for each one, call the function you gave it (lambda here). It reads files in parallel threads.
- **preprocess**() a custom function takes one CSV line and parses it. It uses tf.io.**decode_csv**() which returns a list of scalar tensors, so we need to tf.**stack**() all tensors except the last one (the target) into a 1D array

Prefetching

- By calling **prefetch**(1) at the end, we create a dataset that will do its best to always be one batch ahead (working in parallel to get the next batch ready)
- We can exploit multiple cores in the CPU and hopefully make preparing one batch of data shorter than the training on the GPU → GPU will be almost 100% utilized
- **Tips:** If you plan to buy a GPU card, look for its **memory bandwidth** (the number of GB of data it can I/O of its memory per sec) besides processing power and memory size



Using Datasets with tf.keras

Now let's build a powerful input pipelines using the Data API:

```
train_set = csv_reader_dataset(train_filepaths, repeat=None)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1),
])
```

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
batch_size = 32
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set)
```

Description of methods in the Dataset class

- apply() — Applies a transformation function to this dataset.
- as_numpy_iterator() — Returns an iterator which converts all elements of the dataset to numpy.
- batch() — Combines consecutive elements of this dataset into batches.
- cache() — Caches the elements in this dataset.
- concatenate() — Creates a `Dataset` by concatenating the given dataset with this dataset.
- element_spec() — The type specification of an element of this dataset.
- enumerate() — Enumerates the elements of this dataset.
- filter() — Filters this dataset according to `predicate`.
- flat_map() — Maps `map_func` across this dataset and flattens the result.
- from_generator() — Creates a `Dataset` whose elements are generated by `generator`.
- from_tensor_slices() — Creates a `Dataset` whose elements are slices of the given tensors.
- from_tensors() — Creates a `Dataset` with a single element, comprising the given tensors.
- interleave() — Maps `map_func` across this dataset, and interleaves the results.
- list_files() — A dataset of all files matching one or more glob patterns.
- map() — Maps `map_func` across the elements of this dataset.
- options() — Returns the options for this dataset and its inputs.
- padded_batch() — Combines consecutive elements of this dataset into padded batches.
- prefetch() — Creates a `Dataset` that prefetches elements from this dataset.
- range() — Creates a `Dataset` of a step-separated range of values.
- reduce() — Reduces the input dataset to a single element.
- repeat() — Repeats this dataset so each original value is seen `count` times.
- shard() — Creates a `Dataset` that includes only 1/`num_shards` of this dataset.
- shuffle() — Randomly shuffles the elements of this dataset.
- skip() — Creates a `Dataset` that skips `count` elements from this dataset.
- take() — Creates a `Dataset` with at most `count` elements from this dataset.
- unbatch() — Splits elements of a dataset into multiple elements.
- window() — Combines (nests of) input elements into a dataset of (nests of) windows.
- with_options() — Returns a new `tf.data.Dataset` with the given options set.
- zip() — Creates a `Dataset` by zipping together the given datasets.

TFRecord Format

- **CSV files**, which are common and convenient, are not really efficient, and do not support large or complex data structures (images and audio) very well.
- **TFRecord** is a TensorFlow format for storing large amount of data efficiently.
- It is a binary format containing a sequence of binary records of varying sizes.
- You can create TFRecord file using `tf.io.TFRecordWriter`:

```
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")
```

```
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)
```

Protocol Buffer

- TFRecord files usually contain ***serialized*** protocol buffers (called **protobuf**)
- Protobuf is a portable, and extensible binary format developed at Google
- It is widely used, in particular gRPC (Google's remote procedure call system)
- It is defined using a simple language:

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

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
"Al"
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com")  # add an email address
>>> s = person.SerializeToString()  # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(s)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

TensorFlow ProtoBuf

The main protobuf typically used in TFRecord is the Example protobuf, represent one instance in a dataset:

```
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; };
```

```python
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())
```

Parsing Examples

The following code defines a description dictionary, then iterates over the TFRecord dataset and parses the serialized Example protobuf:

```
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)
```

```
1 parsed_example
```

```
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor at 0x7fa77094b490>,
 'id': <tf.Tensor: shape=(), dtype=int64, numpy=123>,
 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Alice'>}
```

```
1 parsed_example["emails"].values[0]
```

```
<tf.Tensor: shape=(), dtype=string, numpy=b'a@b.com'>
```

Handling Images in TFRecord

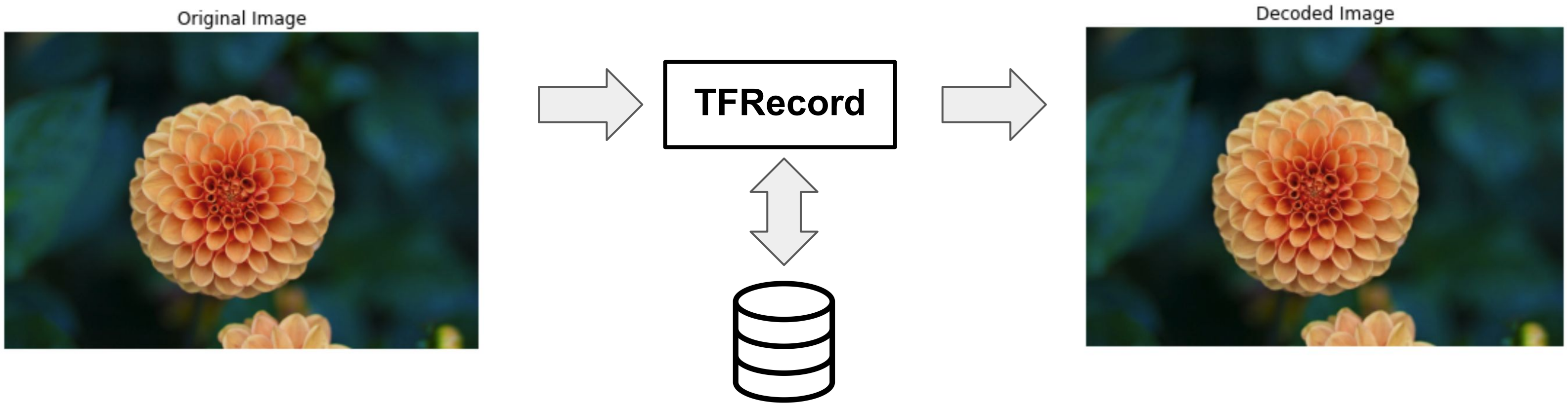
Encoding Image

```python
from sklearn.datasets import load_sample_images
img = load_sample_images()["images"][1]

data = tf.io.encode_jpeg(img)
example_with_image = Example(features=Features(feature={
    "image": Feature(bytes_list=BytesList(value=[data.numpy()]))}))
serialized_example = example_with_image.SerializeToString()
# then save to TFRecord
```

Decoding Image

```python
feature_description = { "image": tf.io.VarLenFeature(tf.string) }
example_with_image = tf.io.parse_single_example(serialized_example, feature_description)
decoded_img = tf.io.decode_image(example_with_image["image"].values[0])
```



Handling Lists of Lists w/ SequenceExample

- Suppose we want to classify text documents, each may consist of a list of sentences, where each sentence is a list of words.
- There may be some contextual data as well (ie. author, title, date)
- We can use SequenceExample protobuf:

```
FeatureList = tf.train.FeatureList
FeatureLists = tf.train.FeatureLists
SequenceExample = tf.train.SequenceExample

context = Features(feature={
    "author_id": Feature(int64_list=Int64List(value=[123])),
    "title": Feature(bytes_list=BytesList(value=[b"A", b"desert", b"place", b"."])),
    "pub_date": Feature(int64_list=Int64List(value=[1623, 12, 25]))
})

content = [["When", "shall", "we", "three", "meet", "again", "?"],
           ["In", "thunder", ",", "lightning", ",", "or", "in", "rain", "?"]]
comments = [["When", "the", "hurlyburly", "'s", "done", "."],
            ["When", "the", "battle", "'s", "lost", "and", "won", "."]]

def words_to_feature(words):
    return Feature(bytes_list=BytesList(value=[word.encode("utf-8")
                                               for word in words]))

content_features = [words_to_feature(sentence) for sentence in content]
comments_features = [words_to_feature(comment) for comment in comments]

sequence_example = SequenceExample(
    context=context,
    feature_lists=FeatureLists(feature_list={
        "content": FeatureList(feature=content_features),
        "comments": FeatureList(feature=comments_features)
    }))
```

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

TensorFlow Datasets (TFDS) Project

The TFDS Project makes it very easy to download common datasets.

The list includes image datasets, text datasets, audio and video datasets

You can download the data you want and return the data as train set and test set (more in the module discussion)

You can then pass the dataset directly to your deep learning model!

```
import tensorflow_datasets as tfds

datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]
```

```
plt.figure(figsize=(6,3))
mnist_train = mnist_train.repeat(5).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    for index in range(5):
        plt.subplot(1, 5, index + 1)
        image = images[index, ..., 0]
        label = labels[index].numpy()
        plt.imshow(image, cmap="binary")
        plt.title(label)
        plt.axis("off")
    break # just showing part of the first batch
```

Recap

- Although you may feel that this video is a bit far from the abstract beauty of neural networks, Deep Learning often involves large amount of data.
- Knowing how to load and preprocess data efficiently is a **crucial** skill to have.
- *You are strongly encourage to **play with the codes** in the **tutorial***

Next, we will look into processing **categorical** and **textual data!**